A Domain Specific Embedded Language in C++ for lowest-order methods for diffusive problem on general meshes

Jean-Marc Gratien

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel :

Présentée par

Jean-Marc GRATIEN

Thèse dirigée par **Christophe Prud'Homme**
et codirigée par **Jean-François Mehaut et Daniele Di Pietro**

préparée au sein **département Informatique Scientifique de l'IFPEN**

A DSEL in C++ for lowest-order methods for diffusive problem on general meshes

Thèse soutenue publiquement le ,
devant le jury composé de :

Monsieur Denis Barthou
Professeur des universités, INRIA Bordeaux, Président
Monsieur David Hill
Professeur des universités, ISIMA, Rapporteur
Madame Marie Rognes
Chef du Département Biomedical Computing, SIMULA, Rapporteur
Monsieur Joël Falcou
Maître de conférences, LRI, Examinateur
Monsieur.Christophe Prud'Homme
Professeur des universités, IRMA, Directeur de thèse
Monsieur Jean-François Méhaut
Professeur des universités, UJF, Co-Directeur de thèse
Monsieur Daniele Di Pietro
Professeur des universités, Université Montpellier 2, Co-Directeur de thèse

A Domain Specific Embedded Language in C++ for lowest-order methods for diffusive problem on general meshes

J.-M. Gratien

December 18, 2013

Remerciements

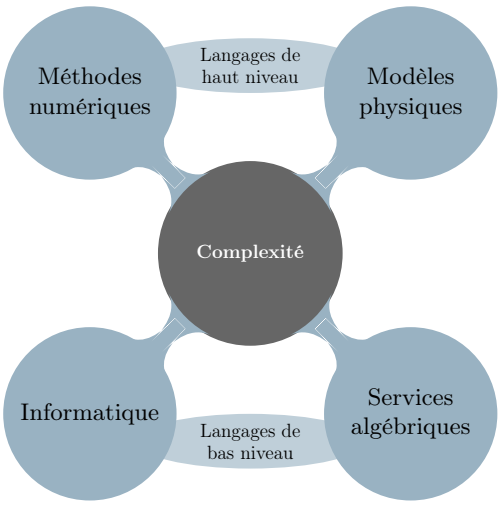
Je dédie mes travaux de thèse à Jonathan, mon dernier fils qui vient d'avoir un an.

Les travaux présentés dans ce rapport n'auraient pu être effectués sans le soutien de ma compagne Longmin, de mes enfants, Jérémie, Célia, William et Jonathan, de mes frères Jean-Noël et Jean-Marie, et de mes parents.

Je tiens à remercier mon directeur Van Bui Tran et mon chef de département Pierre Fery-Forgues qui m'ont donné cette opportunité rare qui est de pourvoir réaliser ces travaux de recherche pendant mon activité professionnelle à IFPEN. Je remercie également Christophe Prud'homme mon directeur de thèse qui s'est tout de suite intéressé au sujet sur lequel je voulais travailler et qui y a contribué de façon décisive par la pertinence de ses remarques et de ses indications. Je remercie Daniele Di-Pietro mon promoteur de thèse qui m'a initié au monde des méthodes numériques avancées pour la résolution d'EDPs et Jean-François Méhaut pour sa contribution dans le domaine de l'informatique et des nouvelles architectures hétérogènes. Je remercie en fin tous mes collègues qui m'ont soutenu et en particulier Ani, Thomas, Youssef, Raphaël, Laurent, Francis, Anthony,... pour leur relecture de ce rapport, leur remarques et leur corrections.

Résumé

Les simulateurs industriels deviennent de plus en plus complexes car ils doivent intégrer à la fois des modèles physiques complets et des méthodes de discrétisation évoluées tout en préservant de bonnes performances. Leur mise au point nécessite donc de gérer de manière efficace (i) la complexité des modèles physiques sous-jacents, souvent exprimés sous la forme de systèmes d'Equations aux Dérivées Partielles (EDPs). Un aspect important est la possibilité de développer ces modèles de manière évolutive dans le but de prolonger la vie d'un code ; (ii) la complexité des méthodes numériques utilisées, dont l'évolution accompagne de nouveaux besoins qui émergent au cours du temps. C'est le cas, par exemple, des méthodes d'ordre bas développées dans le cadre de la simulation de bassins pour traiter les maillages généraux issus de la géomodélisation ; (iii) la complexité des services numériques de bas niveau (gestion du parallélisme, de la mémoire, des interconnexions, GP-GPU) nécessaires pour tirer parti des architectures *hardware* modernes. La difficulté principale est ici de permettre l'évolution de ces services sans remettre en question les autres parties du code ; (iv) la complexité liée aux langages informatiques, dont l'évolution doit être maîtrisée sous peine d'obsolescence du code. Tous ces requis doivent être remplis pour bénéficier pleinement des architectures massivement parallèles et hierarchiques. Cependant, les méthodologies et technologies associées deviennent de plus en plus sophistiquées, et leur maîtrise demande des compétences dont un physicien ou un numericien ne dispose en général pas. Idéalement, la complexité liée aux modèles physiques et aux méthodes numériques se gère mieux par des langages de haut niveaux, qui permettent de cacher les détails informatiques. En revanche, l'efficacité des composantes de bas niveau demande un accès directe aux spécificités *hardware*.



Une réponse partielle au problème est aujourd'hui fournie par des plate-formes qui proposent des outils avancés pour gérer de façon transparente la complexité liée au parallélisme. C'est le cas, par exemple, de la plate-forme `Arcane` [69], co-développée par le CEA et IFP Energies nouvelles, qui propose une interface utilisateur donnant accès aux structures de maillage et d'algèbre linéaire. Cependant, de telles plate-formes n'offrent effectivement qu'une réponse partielle au problème, car elles ne gèrent que la complexité du *hardware* et les services numériques de bas niveau

comme l'algèbre linéaire. Dans le contexte des méthodes Eléments Finis (EF), l'existence d'un cadre mathématique unifié a permis d'envisager des outils qui permettent d'aborder aussi la complexité issue des méthodes numériques et celle liée aux problèmes physiques. C'est le cas, par exemple, de projets tels que `Freefem++`[70], `Getdp`[8], `Getfem++`[9], `Sundance`[16], `Feel++`[86] et `Fenics`[81], qui proposent des langages de haut niveau inspirés par la formulation mathématique. Dans ce cas, le langage est à l'interface entre le mathématicien ou le physicien qui l'utilise et l'informaticien qui s'occupe de le traduire en instructions et algorithmes de bas niveau.

Ce travail vise à étendre ce genre d'approche aux méthodes d'ordre bas pour des systèmes d'EDPs actuellement étudiés dans le cadre des applications en géomodélisation. Ces méthodes constituent un sujet de recherche de pointe en analyse numérique, et elles posent encore de nombreux défis. La motivation principale pour l'étude de ces méthodes est de gérer des maillages généraux. Les premiers essais d'extension de la méthode Volumes Finis (FV) classique à des maillages non orthogonaux dans le contexte de la simulation de réservoirs sont dus à Aavatsmark, Barkve, Bøe et Mannseth [22, 23, 24, 25] et Edwards et Rogers [60, 61]. L'idée de base consiste à remplacer le flux numérique à deux points par une version multi-points pouvant dépendre des valeurs de la solution discrète dans d'autres mailles que celles qui partagent la face. Cependant, si l'introduction des flux multi-points résout le problème de la consistance sur des maillages non orthogonaux, elle ne garantit pas la stabilité de la méthode résultante. Une possible solution aux problèmes de stabilité est fournie alors par les méthodes de Différences Finies Mimétiques (DFM) [42, 41] et par les méthodes Volumes Finis Mixtes/Hybrides (VFMH) [58, 64, 59]. Dans les deux cas, l'idée de base consiste à ajouter des inconnues de faces et à concevoir la méthode en se basant sur la formulation variationnelle plutôt que sur un bilan maille par maille. Le défaut principal de ces méthodes est l'absence d'un cadre suffisamment général permettant son extension à des problèmes différents. Une réponse possible à ce problème a été fournie récemment par les travaux Di Pietro [51, 49, 52], où on introduit une nouvelle classe de méthodes d'ordre bas inspirée par les éléments finis non conformes ; voir aussi Di Pietro et Gratien [57]. Cette formulation permet d'exprimer dans un cadre unifié les schémas VF multi-points et les méthodes DFM/VFMH, et elle s'étend à de nombreux problèmes en mécanique des fluides et des solides.

Ce nouveau cadre fournit la base pour développer des concepts informatiques proche des concepts mathématiques. Plus précisement, nous avons mis au point un langage spécifique (DSEL, *Domain Specific Embedded Language*) en `C++` qui permet aux physiciens ou aux numériciens de développer des applications avec un haut niveau d'abstraction, cachant la complexité des méthodes numériques et des services bas niveau garanties de haute performances. L'objectif d'un tel langage est notamment de permettre le prototypage rapide de codes industriels ou de recherche. La syntaxe et les techniques utilisées sont inspirée par celles de `Feel++` [86], mais le *back-end* est adapté aux méthodes numériques évoquées dans le paragraphe precédent. Parmi les différences majeures par rapport aux méthodes EF, on remarquera, en particulier, l'impossibilité d'utiliser une construction basée sur un élément de référence au sens de Ciarlet et une mappe géometrique. Ceci nous a amené à introduire de nouveaux concepts permettant de gérer les inconnues au niveau global de manière efficace.

Le DSEL a été développé à partir de la plate-forme `Arcane`[69], et embarqué dans le `C++`. Cette approche, partagée par des projets tels que `Feel++`[86] et `Sundance`[16] présente plusieurs avantages par rapport au développement d'un simple langage. Plus précisément, cela (i) évite la construction du compilateur et permet de bénéficier du paradigme génératif du `C++` avec vérification syntaxique au moment de la compilation ; (ii) permet d'utiliser d'autres librairies, et dans notre cas la mise en œuvre proposée se fonde, en particulier, sur de nombreux outils fournis par les bibliothèques `Boost C++` ; (iii) donne la possibilité de bénéficier des mécanismes d'optimisation des compilateurs `C++`, ce qui permet d'envisager des cas plus complexes et de plus grande taille que dans le cas des langages interprétés ; (iv) permet une approche multi-paradigme (programmation orienté objet, fonctionnelle, générique, meta-programmation) adaptée aux calculs scientifiques. En outre certaines extensions du nouveau standard `C++11` (le mot clé `auto`, les functions lambda, etc)

rendent le C++ enfin compétitif du point de vue de son accessibilté en terme de syntaxe par rapport à des langages tels que Python ou Ruby utilisés respectivement dans FreeFem++[70] et Fenics[81], tout en préservant ses qualités de performances.

Les techniques de DSEL sont basées sur les quatres ingrédients clés suivant : (i) la méta-programmation qui consiste à écrire des programmes qui transforment des types à la compilation ; (ii) la programmation générique qui consiste à écrire des composants génériques constitués de programmes abstraits avec des types génériques ; (iii) la programmation générative qui consiste a générer des programmes concrêts en créant des types concrêts avec de la méta-programmation, utilisés dans des programmes abstraits de composants génériques ; (iv) et finalement des techniques d' "expression template" qui consistent à représenter des problèmes et des méthodes de résolution avec des expresssions structurées en arbre et à utiliser des outils pour décrire ces expressions, les parser et les évaluer. Toutes ces techniques permettent de représenter un problème et sa méthode de résolution avec une expression. A la compilation, cette expression peut être parsée, analysée pour générer un programme concrêt en selectionnant des composants génériques, les liant et les assemblant. L'exécution du programme consiste à évaluer l'expresssion et à exécuter des bouts de code sélectionnés pour construire au final un système linéaire que l'on peut résoudre pour trouver la solution du problème. Toutes ces techniques ont été mises au point dans la deuxième partie des années 90 [28, 34, 92], et ont été largement utilisées dans des bibliothèques comme blitz, MTL dès les première années 2000. Elles ont été étendues aux méthodes de types Eléments Finis et aux Equations aux Dérivées Partielles par Feel++[84] en 2005. Elles ont atteint une grande maturité dans des projets à caractères généraux comme les projets Spirit[15], Phoenix[13] et sont maintenant diffusées dans des bibliothèques comme Boost.MPL pour la méta-programmation et Boost.Proto pour la mise au point de DSEL. Cette dernière bibliothèque conçue par Niebler [83] est d'ailleurs en quelque sorte un DSEL en C++ pour mettre au point des DSELs. Utilisée notamment dans les projets Phoenix, NT2[12] et Quaff[14], elle propose un ensemble d'outils pour mettre au point un langage, parser et introspecter des expressions puis générer des algorithmes avec des structures bas niveau basées sur l'évaluation et la transformation d'expressions. Nous avons mis au point notre DSEL à l'aide de ces outils puis l'avons validé sur divers problèmes académiques non triviaux tels que des problèmes de diffusion hétérogène et le problème de Stokes.

Dans un deuxième temps, dans le cadre du projet ANR HAMM (Hybrid Architecture and Multiscale Methods), nous avons validé notre approche en complexifiant le type de méthodes abordées et le type d'architecture *hardware* cible pour nos programmes. Nous avons étendu le formalisme mathématique sur lequel nous nous basons pour pouvoir écrire des méthodes multi-échelle puis nous avons enrichi notre DSEL pour pouvoir implementer de telles méthodes.

Afin de pouvoir tirer partie de façon transparente des performances de ressources issues d'architectures hybrides proposant des cartes graphiques de type GP-GPU, nous avons mis au point une couche abstraite proposant un modèle de programmation unifié qui permet d'accéder à differents niveaux de parallélisme plus ou moins fin en fonction des spécificités de l'architecture matérielle cible. Nous avons validé cette approche en évaluant les performances de cas tests utilisant des méthodes multi-échelle sur des configurations variés de machines hétérogènes.

Pour finir nous avons implémenté des applications variées de type diffusion-advection-réaction, de Navier-Stokes incompressible et de type réservoir. Nous avons validé la flexibilité de notre approche et la capacité qu'elle offre à appréhender des problèmes variés puis avons étudié les performances des diverses implémentations.

Pour poursuivre nos travaux, nous envisageons : (i) d'étendre notre DSEL pour prendre en compte des formulations non linéaires en introduisant les concepts de dérivées de Fréchet[82] qui permettent de cacher la complexité de la gestion des dérivées analytiques ; (ii) d'adresser d'autres domaines d'application tel que la poro-mécanique ; (iii) d'étudier la possiblité d'étendre notre approche pour décrire des méthodes Volumes Finies pour des problèmes d'advection.

Contents

Chapter 1

Context and motivation

Industrial simulation software has to manage: *(i)* the complexity of the underlying physical models, usually expressed in terms of a PDE system completed with algebraic closure laws, *(ii)* the complexity of numerical methods used to solve the PDE systems, and finally *(iii)* the complexity of the low level computer science services required to have efficient software on modern hardware. Robust and effective finite volume (FV) methods as well as advanced programming techniques need to be combined in order to fully benefit from massively parallel architectures (implementation of parallelism, memory handling, design of connections). Moreover, the above methodologies and technologies become more and more sophisticated and too complex to be handled only by physicists. Nowadays, this complexity management (figure 1) becomes a key issue for the development of scientific software.

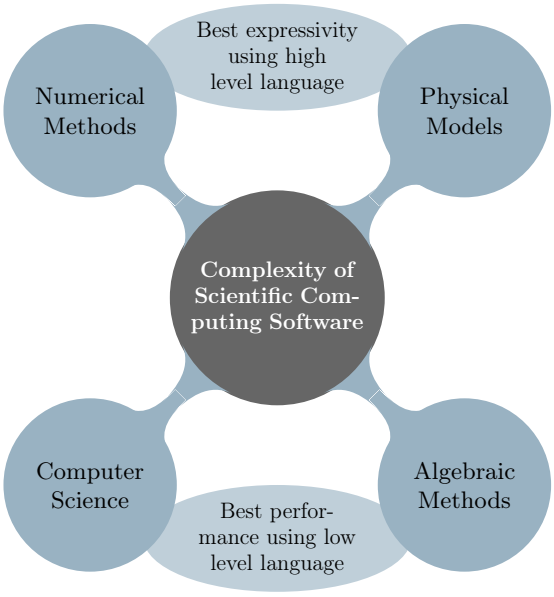


Figure 1.1: Complexity management

1.1 Complexity management

Some frameworks already offer a number of advanced tools to deal with the complexity related to parallelism in a transparent way. Hardware complexity is hidden and low level algorithms which need to deal directly with hardware specificity, for performance reasons, are provided. They often

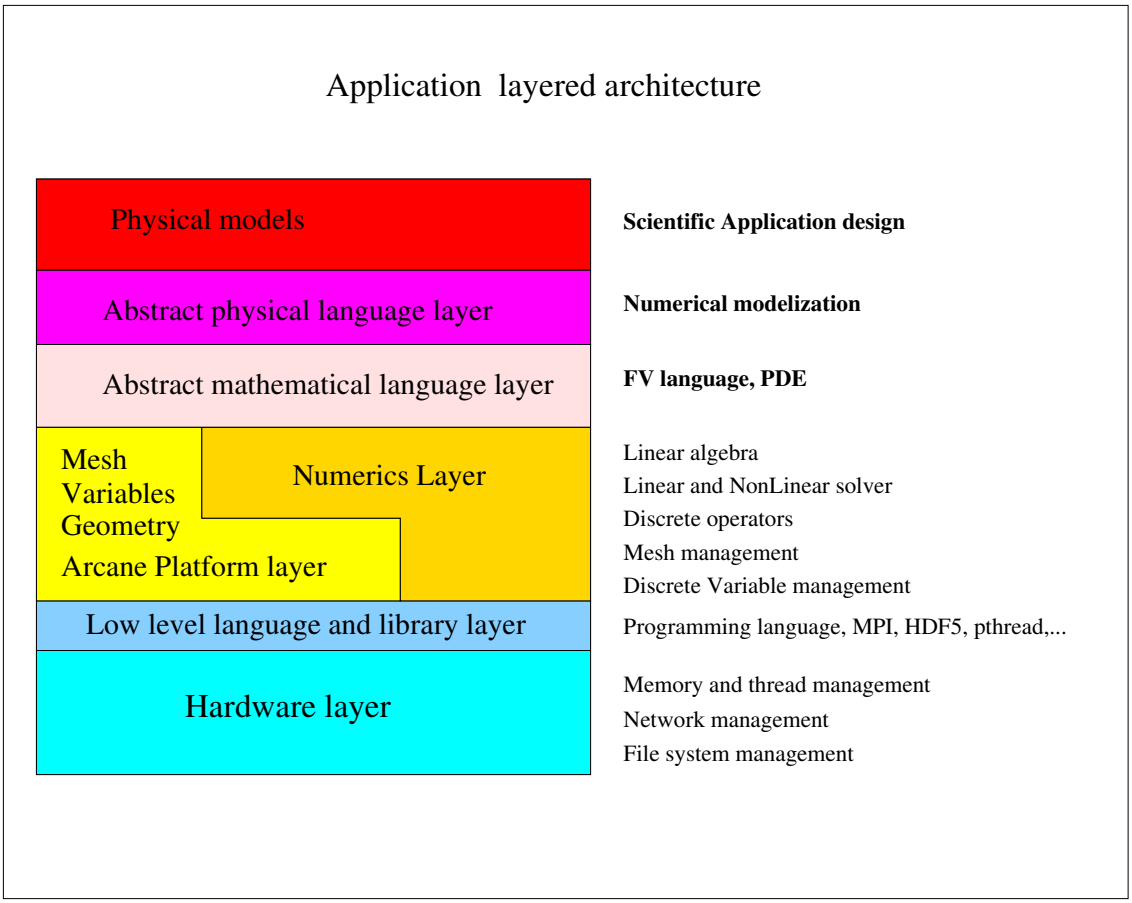


Figure 1.2: Layer Architecture.

offer services to manage mesh data services and linear algebra services which are key elements to have efficient parallel software.

Layer architecture, see figure (1.2), helps to establish a hierarchy between the different complexity levels and helps to limit the transfer of complex technical information between neighbouring layers. However, all these frameworks often provide only partial answers to the problem as they only deal with hardware complexity and low level numerical complexity like linear algebra. High level complexity related to discretization methods and physical models lack tools to help physicists to develop complex applications. New paradigms for scientific software must be developed to help them to seamlessly handle the different levels of complexity so that they can focus on their specific domain. Generative programming, component engineering and domain-specific languages (either DSL or DSEL) are key technologies to make the development of complex applications easier to physicists, hiding the complexity of numerical methods and low level computer science services. These paradigms allow to write code with a high level expressive language and take advantage of the efficiency of generated code for low level services close to hardware specificities (figure 1.1). In scientific computing, these paradigms were first applied in linear algebra framework like `blitz`, `MTL4` or `Eigen`. But in in the domain of numerical algorithms to solve partial differential equations, their application has been up to now limited to Finite Element (FE) methods, for which a unified mathematical framework has been existing for a long time. Such kinds of DSL have been developed for finite element or Galerkin methods in projects like `Freefem++`[70], `Getdp`[8], `Getfem++`[9], `Sundance`[16], `Feel++`[86] and `Fenics`[81]. The advantages that provide such languages have no more to be proved [84, 85]. They are used for various reasons, teaching purposes,

design of complex problems or rapid prototyping of new methods, schemes or algorithms, the main goal being always to hide technical details behind software layers and provide only the relevant components required by the user or programmer.

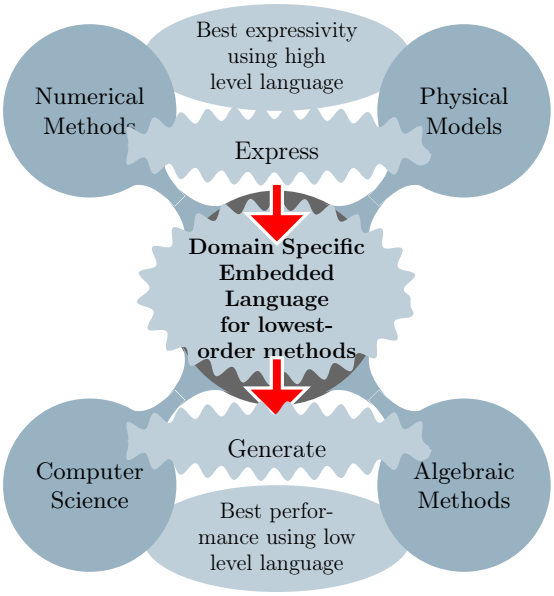


Figure 1.3: DSEL and generative programming

1.2 Computational frameworks overview

In scientific computing, frameworks have emerged to help physicists and numericians to design their applications as soon as the performance issues of algebraic algorithms became crucial while the consideration of hardware specificities like vectorisation optimisation started to be essential to get performance. The first frameworks were concerning the algebraic layer with matrix and vector operations. This low level layer for numericians is one of the most CPU consuming and very early optimized generic libraries have been developed. Later some projects introduced meshes and useful structures to manage both algebraic and geometrical objects. Finally modern frameworks tried to bring solutions at a higher level to design numerical methods to solve partial differential equations.

1.2.1 Linear and non linear algebra framework

After the success of the `Blas`, `Lapack` and `Scalapack` libraries, various projects have been developed to design frameworks providing efficient and complex methods to solve linear and non linear systems for large parallel with distributed memory architectures. Among them `Trilinos`[72] and `Petsc`[37] are dynamic projects, famous for their rich libraries providing a great number of parallel algorithms and services that enable to take advantage easily of new parallel clusters.

More recently projects like `MTL4`[11] or `Eigen`[5] have introduced the generative paradigm in their linear packages and developed DSELs to help numericians to develop complex algorithms based on matrices-vectors operations without taking care of low level optimizations like for instance sse or openmp optimizations. These frameworks provide a much more friendly to use interface than the previous ones while they keep being efficient on new hardware platforms.

1.2.2 Frameworks to solve partial differential equations systems

With the trend of large parallel clusters with distributed memory, frameworks have naturally emerged above the linear algebraic layer to make easier the development of algorithms to solve partial differential equations. They usually introduce a layer to abstract the management of meshes with partitioner, import/export services, low level communication services, ... Among such those frameworks:

- **Arcane Platform**[69] is a parallel C++ framework, co-developed by CEA (Commissariat à l'Energie Atomique) and IFP New Energy, designed to develop applications based on 1D, 2D, 3D unstructured grids. In the Arcane component based architecture, functionalities are provided via components, called services in Arcane terminology. The framework is extendable via a dynamic plug-in mechanism which makes Arcane highly adaptable and not tied to a specific implementation. It provides services to manage meshes, mesh elements, groups of mesh elements, manage discrete variables representing discrete fields on mesh elements, and extra low level services to synchronize discrete variables and to manage parallelism and network communication between processors and IO services. A linear algebra layer also developed above the platform, provides a unified way to handle standard parallel linear solver packages such as Petsc, Hypre, MTL4, UBlas and IFPSolver, an in house linear solver package.

- **DUNE**[4], the Distributed and Unified Numerics Environment is a modular framework for solving partial differential equations (PDEs) with grid-based methods like Finite Elements (FE), Finite Volumes (FV), and also Finite Differences (FD). Its efficient implementation is based on the use of generic programming techniques. It enables the reuse of existing finite element packages in particular the finite element codes UG, ALBERTA, and ALUGrid. This framework is used in particular in the OPM project (Open Porous Media) developing the free simulator $DuMu^x$, DUNE for Multi-{Phase, Component, Scale, Physics, ...} flow and transport in porous media.

Some frameworks are dedicated to solve partial differential equations with methods based on variational formulation and provide moreover DSELs with high level abstractions closed to the mathematical formalism. Among them:

- **Sundance**[16] is a library with automatic differentiation technics based on in-place Frechet differentiation of symbolic objects allowing optimization, uncertainty quantification, and adaptive error control.

- The **FEniCS Project**[6] provides a collection of open-source components. Among them, as written in the wikipedia project page [7], there are `UFL` (Unified Form Language) providing a domain-specific language embedded in Python, `FIAT` (Finite element Automatic Tabulator), a "Python module for generation of arbitrary order finite element basis functions on simplices", `FFC` (FEniCS Form Compiler), a compiler of `UFL` code generating `UFC` code (Unified Form-assembly Code) with "low-level functions in C++ for evaluating and assembling finite element variational forms", `Instant`, a "Python module for inlining C and C++ code in Python" and `DOLFIN`, a "C++/Python library providing data structures and algorithms for finite element meshes, automated finite element assembly, and numerical linear algebra".

- **FEEL++**[84, 85, 86] provides a DSEL embedded in C++, linked with established libraries (`Petsc`, `Trilinos` for linear and non-linear solvers, `Gmsh` and `Metis` for mesh services, `MPI` for parallelism,...).

1.3 Mathematical methods to solve partial differential equations overview

To solve partial differential equations, the finite element methods (FEM), the finite volume methods (FVM) and the finite difference methods (FDM) belong to the family of methods most widely used.

1.3.1 Finite Difference Methods

The Finite-difference methods, developed for a long time, consist in approximating the solutions to differential equations using finite difference equations to approximate derivatives. Usually based on regular meshes, they can be developed with algebraic frameworks like `Petsc`, `Hypre` and `Trilinos` that provides to their matrix vector interfaces helper tools to map regular meshes indexes to algebraics object indexes.

1.3.2 Finite Element Methods

The Finite Element Methods, and its other versions the generalized finite element method (GFEM), the extended finite element method (XFEM), the spectral finite element method (SFEM), the meshfree finite element method or the discontinuous Galerkin finite element method (DGFEM) belong to the class of methods called Generalized Galerkin methods widely used to solve PDEs. These methods consist in converting the continuous formulation of the PDEs into a discrete formulation using the method of variation of parameters to a function space, by converting the equations system to a weak formulation. A unified mathematical formalism to describe these methods existing for a long time[46], various frameworks dedicated to them are widely used and have nowadays reached a good level of maturity. They generally provide a user-friendly front-end in the form of a Domain Specific Language (DSL) possibly embedded in a general purpose, high-level hosting language (Domain Specific Embedded Language or DSEL). They are often based on unstructured meshes and are well known for the efficiency of their higher-order version hp-FEM.

1.3.3 Finite Volume Methods

Finite Volume methods are a kind of generalisation of the finite difference method to general meshes. This method consists in converting surface integrals with divergence term to volume integrals, using the divergence theorem. These terms are then evaluated as fluxes at the surfaces of each finite volume. These methods, naturally conservative, is often used by physicists. They are employed in industrial applications where computational cost is a crucial issue. In this context, the use of general polyhedral, possibly nonconforming meshes commends itself for a number of reasons. To cite a few: (i) remeshing can be avoided or postponed in problems that involve mesh deformation — e.g. in sedimentary basin modeling non-standard elements and nonconformities can appear due to the erosion of geological layers; — (ii) the number of degrees of freedom can be reduced by aggregative coarsening techniques — cf. [38] for an application in the context of discontinuous Galerkin (dG) methods; — (iii) geometrical features can be represented more accurately without unduly increasing the number of mesh elements. Handling general polyhedral meshes requires numerical schemes that possess the usual properties of stability and consistency. In the context of cell centered finite volume methods, a popular way to achieve consistency on general polyhedral meshes is provided by Multipoint Finite Volume schemes independently introduced by Aavatsmark, Barkve, Bøe and Mannseth [24] and Edwards and Rogers [61]. The main advantage of multipoint schemes is that they can be easily fitted into existing simulators based on standard finite volume schemes. A major drawback is their lack of stability in some configurations. Two ways of overcoming this difficulty by designing discretizations based on the variational formulation of the problem and featuring cell- and face-unknowns have been proposed by Brezzi, Lipnikov and coworkers [42, 41] (Mimetic Finite Difference methods) and by Droniou and Eymard [58] (Mixed/Hybrid Finite Volume methods). In this context, Eymard, Gallouët

and Herbin [65] have shown that face unknowns can be selectively used as Lagrange multipliers to enforce flux continuity, or eliminated using a consistent interpolator (SUSHI scheme). More generally, this point of view leads to the notion that the discretization method can be locally adapted to the features of the problem. A different approach based on the analogy between lowest order methods in variational formulation and discontinuous Galerkin methods has been proposed in [49, 52, 53] (Cell Centered Galerkin methods). The key advantage of this approach is that it largely benefits from the well-established theory for discontinuous Galerkin methods [54]. When it comes to numerical performance, recent benchmarks [71, 66] have pointed out that the choice of the scheme for a given problem should be driven by multiple factors including, e.g., the features of the problem itself (heterogeneity, presence of convection), the computational mesh (which may result from an upstream modeling process), and the required precision.

Contrary to FE methods up to recently the lack of unified mathematical formalism to describe all lowest order methods leads to the fact that no serious framework covering a wide range of these methods has emerged. In this respect, there is an increasing urge to dispose of libraries and applications based on similar experiences in the context of Finite Element (FE) methods.

1.4 Proposition

A new consistent unified mathematical frame has recently emerged and allows a unified description of a large family of lowest-order methods [31, 29, 49]. This framework allows then, as in FE methods, the design of a high level language inspired from the mathematical notation, that could help physicists to implement their application writing the mathematical formulation at a high level, hiding the complexity of numerical methods, while low level computer science services ensure the efficiency. We propose to develop a language based on that frame, embedded in the C++ language. This approach, used in projects like `Feel++` or `Sundance` has several advantages over generating a specific language. Embedded in the C++ language, (i) it avoids the compiler construction complexities, taking advantage of the generative paradigm of the C++ language and allowing grammar checking at compile time; (ii) it allows to use other libraries concurrently which is often not the case for specific languages, our implementation heavily relies, in particular, on the tools provided by the `boost` library; (iii) it exploits the optimization capabilities of the C++ compiler, thereby allowing to tackle large study cases which is not possible with interpreted language; (iv) it allows multiple paradigm programming as meta-programming, object oriented, generic and functional programming. New concepts provided by the standard C++11 (the keyword `auto`, lambda functions, ...), make C++ very competitive as its syntax becomes comparable to that of interpreted languages like `Python` or `Ruby` used in projects like `FreeFem++` or `Fenics`, while performance issues remain preserved thanks to compiler optimisations.

In Chapter 2 we present the mathematical framework that enables us to describe a wide family of lowest order methods including multiscale methods based on lowest order methods.

In Chapter 3 we propose a DSEL developed on top of Arcane platform 1.2.2, based on the concepts presented in the unified mathematical frame and on the `Feel++` DSEL. We present the `C++` representations of these mathematical concepts, which are the foundation for the user-friendly interface, the front-end of a specific language which conceals most of the implementation details and allows the numerician to focus on discretization methods. The DSEL is implemented with the `Boost.Proto` library by Niebler [83], a powerful framework to build a DSEL in C++ in a declarative way, which provides a collection of generic concepts and metafunctions that help to design a DSL, its grammar and tools to parse and evaluate expressions. We provide several numerical examples to assess the performance of the proposed approach. We propose an extension of the computational framework to mutiscale methods. We focus on the capability of our approach to handle complex methods based on the basic bricks of the framework and to describe and implement new methods assembling them. We validate our approach with multiscale methods with

several numerical examples.

In Chapter 4 we extend our approach to the runtime system layer providing an abstract layer that enable our DSEL to generate efficient code for heterogeneous architectures. We validate the design of this layer by benchmarking the multiscale method described in §2.5. This method provides a great amount of independent computations and is therefore the kind of algorithms that can take advantage efficiently of new hybrid hardware technology.

Finally in Chapter 5 we benchmark various complex applications and study the performance results of their implementations with our DSEL.

Chapter 2

Mathematical setting

This chapter is largely taken, up to section 2.4, from [48], our article published in Bit Numerical Mathematics. In the last section we turn to the mathematical extension to multiscale problems.

The unified mathematical frame presented in [52, 57] allows a unified description of a large family of lowest-order methods. The key idea is to reformulate the method at hand as a (Petrov)-Galerkin scheme based on a possibly incomplete, broken affine space. This is done by introducing a piecewise constant gradient reconstruction, which is used to recover a piecewise affine function starting from cell (and possibly face) centered unknowns. In this section we briefly present some of these methods, in particular the cell centered Galerkin (ccG) method and the G-method with cell unknowns only and the methods of the mimetic finite difference (MFD) and mixed/hybrid finite volume (MHFV) family with both cell and face unknowns.

2.1 Mesh

Let $\Omega \subset \mathbb{R}^d$, $d \geq 2$, denote a bounded connected polyhedral domain. The first ingredient in the definition of lowest order methods is a suitable discretization of Ω. We denote by $\mathcal{T}_h$ a finite collection of nonempty, disjoint open polyhedra $\mathcal{T}_h = \{T\}$ forming a partition of Ω such that $h = \max_{T\in\mathcal{T}_h} h_T$, with h_T denoting the diameter of the element $T \in \mathcal{T}_h$. Admissible meshes include general polyhedral discretizations with possibly nonconforming interfaces; see Figure 2.1 for an example in $d = 2$. Mesh nodes are collected in the set $\mathcal{N}_h$ and, for all $T \in \mathcal{T}_h$, $\mathcal{N}_T$ contains the nodes that lie on the boundary of T. We say that a hyperplanar closed subset F of $\overline{\Omega}$ is a mesh face if it has positive $(d-1)$-dimensional measure and if either there exist T_1, $T_2 \in \mathcal{T}_h$ such that $F \subset \partial T_1 \cap \partial T_2$ (and F is called an *interface*) or there exist $T \in \mathcal{T}_h$ such that $F \subset \partial T \cap \partial\Omega$ (and F is called a *boundary face*). Interfaces are collected in the set $\mathcal{F}_h^i$, boundary faces in $\mathcal{F}_h^b$ and we let $\mathcal{F}_h \stackrel{\text{def}}{=} \mathcal{F}_h^i \cup \mathcal{F}_h^b$. Moreover, we set, for all $T \in \mathcal{T}_h$,

$$\mathcal{F}_T \stackrel{\text{def}}{=} \{F \in \mathcal{F}_h \mid F \subset \partial T\}. \tag{2.1}$$

Symmetrically, for all $F \in \mathcal{F}_h$, we define

$$\mathcal{T}_F \stackrel{\text{def}}{=} \{T \in \mathcal{T}_h \mid F \subset \partial T\}.$$

The set $\mathcal{T}_F$ consists of exactly two mesh elements if $F \in \mathcal{F}_h^i$ and of one if $F \in \mathcal{F}_h^b$. For all mesh nodes $P \in \mathcal{N}_h$, $\mathcal{F}_P$ denotes the set of mesh faces sharing P, i.e.

$$\mathcal{F}_P \stackrel{\text{def}}{=} \{F \in \mathcal{F}_h \mid P \in F\}. \tag{2.2}$$

For every interface $F \in \mathcal{F}_h^i$ we introduce an arbitrary but fixed ordering of the elements in $\mathcal{T}_F$ and let $\mathrm{n}_F = \mathrm{n}_{T_1,F} = -\mathrm{n}_{T_2,F}$, where $\mathrm{n}_{T_i,F}$, $i \in \{1, 2\}$, denotes the unit normal to F pointing out of

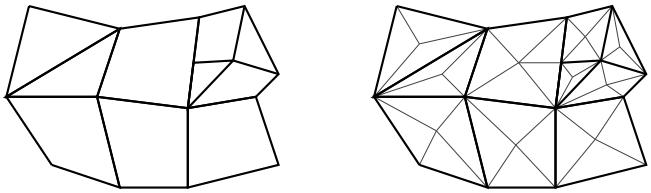


Figure 2.1: *Left.* Mesh $\mathcal{T}_h$ *Right.* Pyramidal submesh $\mathcal{P}_h$

$T_i \in \mathcal{T}_F$. On a boundary face $F \in \mathcal{F}_h^b$ we let n_F denote the unit normal pointing out of Ω. The barycenter of a face $F \in \mathcal{F}_h$ is denoted by $\overline{\mathbf{x}}_F \stackrel{\text{def}}{=} \int_F \mathbf{x}/|F|_{d-1}$.

For each $T \in \mathcal{T}_h$ we identify a point $\mathbf{x}_T \in T$ (the *cell center*) such that T is star-shaped with respect to $\mathbf{x}_T$. For all $F \in \mathcal{F}_T$ we let

$$d_{T,F} \stackrel{\text{def}}{=} \operatorname{dist}(\mathbf{x}_T, F).$$

It is assumed that, for all $T \in \mathcal{T}_h$ and all $F \in \mathcal{F}_T$, $d_{T,F} > 0$ is uniformly comparable to h_T. Starting from cell centers we can define a non degenerate pyramidal submesh of $\mathcal{T}_h$ as follows:

$$\mathcal{P}_h \stackrel{\text{def}}{=} \{\mathcal{P}_{T,F}\}_{T\in\mathcal{T}_h,\, F\in\mathcal{F}_T},$$

where, for all $T \in \mathcal{T}_h$ and all $F \in \mathcal{F}_T$, $\mathcal{P}_{T,F}$ denotes the open pyramid of apex $\mathbf{x}_T$ and base F, i.e.,

$$\mathcal{P}_{T,F} \stackrel{\text{def}}{=} \{\mathbf{x} \in T \mid \exists \mathbf{y} \in F \setminus \partial F,\, \exists \theta \in (0,1) \mid \mathbf{x} = \theta\mathbf{y} + (1-\theta)\mathbf{x}_T\}.$$

Let $\mathcal{S}_h$ be such that

$$\mathcal{S}_h = \mathcal{T}_h \text{ or } \mathcal{S}_h = \mathcal{P}_h. \tag{2.3}$$

For all $k \geq 0$, we define the broken polynomial spaces of total degree $\leq k$ on $\mathcal{S}_h$,

$$\mathbb{P}_d^k(\mathcal{S}_h) \stackrel{\text{def}}{=} \{v \in L^2(\Omega) \mid \forall S \in \mathcal{S}_h,\, v|_S \in \mathbb{P}_d^k(S)\},$$

with $\mathbb{P}_d^k(S)$ given by the restriction to $S \in \mathcal{S}_h$ of the functions in $\mathbb{P}_d^k$ where $\mathbb{P}_d^k$ is the space of polynomial functions in d variables of total degree $\leq k$.

We introduce trace operators which are of common use in the context of nonconforming finite element methods. Let v be a scalar-valued function defined on Ω smooth enough to admit on all $F \in \mathcal{F}_h$ a possibly two-valued trace. To any interface $F \subset \partial T_1 \cap \partial T_2$ we assign two non-negative real numbers $\omega_{T_1,F}$ and $\omega_{T_2,F}$ such that

$$\omega_{T_1,F} + \omega_{T_2,F} = 1,$$

and define the jump and weighted average of v at F for a.e. $\mathbf{x} \in F$ as

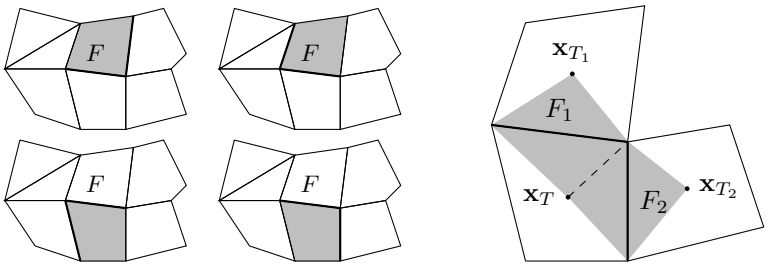
$$[\![v]\!]_F(\mathbf{x}) \stackrel{\text{def}}{=} v|_{T_1} - v|_{T_2}, \qquad \{v\}_{\omega,F}(\mathbf{x}) \stackrel{\text{def}}{=} \omega_{T_1,F} v|_{T_1}(\mathbf{x}) + \omega_{T_2,F} v|_{T_2}(\mathbf{x}). \tag{2.4}$$

If $F \in \mathcal{F}_h^b$ with $F = \partial T \cap \partial\Omega$, we conventionally set $\{v\}_{\omega,F}(\mathbf{x}) = [\![v]\!]_F(\mathbf{x}) = v|_T(\mathbf{x})$. The index ω is omitted from the average operator when $\omega_{T_1,F} = \omega_{T_2,F} = \frac{1}{2}$, and we simply write $\{v\}_F(\mathbf{x})$. The dependence on $\mathbf{x}$ and on the face F is also omitted from both the jump and average trace operator if no ambiguity arises.

2.2 Degrees of freedoms

Let $\mathbb{T}_h \stackrel{\text{def}}{=} \mathbb{R}^{\mathcal{T}_h}$ and $\mathbb{F}_h \stackrel{\text{def}}{=} \mathbb{R}^{\mathcal{F}_h}$. We define the **space of degrees of freedom** (DOFs) as the vector space $\mathbb{V}_h$ such that $\mathbb{V}_h = \mathbb{T}_h$ or $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$. The choice $\mathbb{V}_h = \mathbb{T}_h$ is refered to as a **Cell Centered space of DOFs**, whereas the choice $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$ is refered to as **Hybrid Space of DOFs**.

Elements $v \in \mathbb{V}_h$ are called discrete variables. They often represent in the FV community the piecewise constant functions $\mathbf{v}$ where for $\tau \in \mathcal{T}_h$, v_τ represents the value of the function on τ.



(a) $\mathcal{G}_F$ = L-groups containing the face F (b) L-construction

Figure 2.2: L-construction

2.3 A unified abstract perspective for lowest-order methods

The key idea to gain a unifying perspective is to regard lowest order methods as nonconforming methods based on incomplete broken affine spaces that are defined starting from the vector space $\mathbb{V}_h$ of DOFs. The cell centered space of DOFs $\mathbb{V}_h = \mathbb{T}_h$ corresponds to cell centered finite volume (CCFV) and cell centered Galerkin (CCG) methods, while the hybrid space of DOFs $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$ leads to mimetic finite difference (MFD) and mixed/hybrid finite volume (MHFV) methods. The key ingredient in the definition of a broken affine space is a piecewise constant linear gradient reconstruction $\mathfrak{G}_h : \mathbb{V}_h \to [\mathbb{P}_d^0(\mathcal{S}_h)]^d$ with suitable properties. We emphasize that the linearity of $\mathfrak{G}_h$ is a founding assumption for the implementation discussed in Chapter 3.

Using the above ingredients, we can define the linear operator $\mathfrak{R}_h : \mathbb{V}_h \to \mathbb{P}_d^1(\mathcal{S}_h)$ such that, for all $\mathbf{v}_h \in \mathbb{V}_h$,

$$\forall S \in \mathcal{S}_h,\, S \subset T_S,\, T_S \in \mathcal{T}_h,\, \forall \mathbf{x} \in S, \qquad \mathfrak{R}_h(\mathbf{v}_h)|_S = v_{T_S} + \mathfrak{G}_h(\mathbf{v}_h)|_S \cdot (\mathbf{x} - \mathbf{x}_{T_S}). \tag{2.5}$$

The operator $\mathfrak{R}_h$ maps every vector of DOFs $\mathbf{v}_h \in \mathbb{V}_h$ onto a piecewise affine function $\mathfrak{G}_h(\mathbf{v}_h)$ belonging to $\mathbb{P}_d^1(\mathcal{S}_h)$. Hence, we can define a broken affine space as follows:

$$V_h = \mathfrak{R}_h(\mathbb{V}_h) \subset \mathbb{P}_d^1(\mathcal{S}_h). \tag{2.6}$$

The operator $\mathfrak{R}_h$ is assumed to be injective, so that a bijective operator can be obtained by restricting its codomain. In what follows we show how some common lowest order methods can be interpreted in this terms. To simplify the exposition, we focus on discrete spaces approximating $H_0^1(\Omega)$, i.e., possibly including strongly enforced boundary conditions.

2.4 Examples of gradient operator

In this section we provide a few examples for the gradient operator $\mathfrak{G}_h$ that allow to recover some of the methods listed in the previous section for the following heterogeneous diffusion model problem :

$$\begin{aligned} -\nabla\cdot(\kappa\nabla u) &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega, \end{aligned}$$

with source term $f \in L^2(\Omega)$.

2.4.1 The G-method

As a first example we consider the special instance of CCFV methods analyzed in [29]. A preliminary step consists in introducing the so-called L-construction originally proposed in [26]. The key idea of the L-construction is to use d cell and boundary face values (provided, in this case, by the homogeneous boundary condition) to express a continuous piecewise affine function with

continuous diffusive fluxes. The values are selected using d neighboring faces belonging to a cell and sharing a common vertex. More precisely, we define the set of L-groups as follows:

$$\mathcal{G} \stackrel{\text{def}}{=} \{\mathfrak{g} \subset \mathcal{F}_T \cap \mathcal{F}_P,\, T \in \mathcal{T}_h,\, P \in \mathcal{N}_T \mid \operatorname{card}(\mathfrak{g}) = d\},$$

with $\mathcal{F}_T$ and $\mathcal{F}_P$ given by (2.1) and (2.2) respectively. It is useful to introduce a symbol for the set of cells concurring in the L-construction: For all $\mathfrak{g} \in \mathcal{G}$, we let

$$\mathcal{T}_\mathfrak{g} \stackrel{\text{def}}{=} \{T \in \mathcal{T}_h \mid T \in \mathcal{T}_F,\, F \in \mathfrak{g}\}.$$

Let now $\mathfrak{g} \in \mathcal{G}$ and denote by $T_\mathfrak{g}$ an element $T_\mathfrak{g}$ such that $\mathfrak{g} \subset \mathcal{F}_{T_\mathfrak{g}}$ (this element may not be unique). For all $\mathbf{v}_h \in \mathbb{V}_h$ we construct the function $\xi^\mathfrak{g}_{\mathbf{v}_{\mathbf{h}_h}}$ piecewise affine on the family of pyramids $\{\mathcal{P}_{T,F}\}_{F\in\mathfrak{g},\,T\in\mathcal{T}_\mathfrak{g}}$ such that: (i) $\xi^\mathfrak{g}_{\mathbf{v}_{\mathbf{h}_h}}(\mathbf{x}_T) = v_T$ for all $T \in \mathcal{T}_\mathfrak{g}$ and $\xi^\mathfrak{g}_{\mathbf{v}_{\mathbf{h}_h}}(\overline{\mathbf{x}}_F) = 0$ for all $F \in \mathfrak{g} \cap \mathcal{F}_h^b$; (ii) $\xi^\mathfrak{g}_{\mathbf{v}_{\mathbf{h}_h}}$ is affine inside $T_\mathfrak{g}$ and is continuous across every interface in the group: For all $F \in \mathfrak{g} \cap \mathcal{F}_h^i$ such that $F \subset \partial T_1 \cap \partial T_2$,

$$\forall \mathbf{x} \in F, \qquad \xi^\mathfrak{g}_{\mathbf{v}_{\mathbf{h}_h}}|_{T_1}(\mathbf{x}) = \xi^\mathfrak{g}_{\mathbf{v}_{\mathbf{h}_h}}|_{T_2}(\mathbf{x});$$

(iii) $\xi^\mathfrak{g}_{\mathbf{v}_{\mathbf{h}_h}}$ has continuous diffusive flux across every interface in the group: For all $F \in \mathfrak{g} \cap \mathcal{F}_h^i$ such that $F \subset \partial T_1 \cap \partial T_2$,

$$(\kappa \nabla \xi^\mathfrak{g}_{\mathbf{v}_{\mathbf{h}_h}})|_{T_1} \cdot \mathrm{n}_F = (\kappa \nabla \xi^\mathfrak{g}_{\mathbf{v}_{\mathbf{h}_h}})|_{T_2} \cdot \mathrm{n}_F.$$

For further details on the L-construction we refer to [26, 29]. For every face $F \in \mathcal{F}_h$ we define the set $\mathcal{G}_F$ of L-groups to which F belongs

$$\mathcal{G}_F \stackrel{\text{def}}{=} \{\mathfrak{g} \in \mathcal{G} \mid F \in \mathfrak{g}\}, \tag{2.7}$$

and introduce the set of non-negative weights $\{\omega_{\mathfrak{g},F}\}_{\mathfrak{g}\in\mathcal{G}_F}$ such that $\sum_{\mathfrak{g}\in\mathcal{G}_F} \omega_{\mathfrak{g},F} = 1$. The trial space for the G-method is obtained as follows: (i) let $\mathcal{S}_h = \mathcal{P}_h$ and $\mathbb{V}_h = \mathbb{T}_h$; (ii) let $\mathfrak{G}_h = \mathfrak{G}_h^\mathrm{g}$ with $\mathfrak{G}_h^\mathrm{g}$ such that

$$\forall \mathbf{v}_h \in \mathbb{T}_h,\, \forall T \in \mathcal{T}_h,\, \forall F \in \mathcal{F}_T, \qquad \mathfrak{G}_h^\mathrm{g}(\mathbf{v}_h)|_{\mathcal{P}_{T,F}} = \sum_{\mathfrak{g}\in\mathcal{G}_F} \omega_{\mathfrak{g},F} \nabla \xi^\mathfrak{g}_{\mathbf{v}_{\mathbf{h}_h}}|_{\mathcal{P}_{T,F}}.$$

We denote by $\mathfrak{R}_h^\mathrm{g}$ the reconstruction operator defined as in (2.5) with $\mathfrak{G}_h = \mathfrak{G}_h^\mathrm{g}$ and let $V_h^\mathrm{g} \stackrel{\text{def}}{=} \mathfrak{R}_h^\mathrm{g}(\mathbb{V}_h)$. The G-method of [29] is then equivalent to the following Petrov-Galerkin method:

$$\text{Find } u_h \in V_h^\mathrm{g} \text{ s.t. } a_h^\mathrm{g}(u_h, v_h) = \int_\Omega f v_h \text{ for all } v_h \in \mathbb{P}_d^0(\mathcal{T}_h), \tag{2.8}$$

where $a_h^\mathrm{g}(u_h, v_h) \stackrel{\text{def}}{=} -\sum_{F\in\mathcal{F}_h} \int_F \{\kappa \nabla_h u_h\} \cdot \mathrm{n}_F [\![v_h]\!]$ with ∇_h broken gradient on $\mathcal{S}_h$.

2.4.2 A cell centered Galerkin method

The L-construction is also used to define a trace reconstruction to be used in the CCG method of [50, 49]. More specifically, for all $F \in \mathcal{F}_h^i$ we select one group $\mathfrak{g}_F \in \mathcal{G}_F$ with $\mathcal{G}_F$ defined by (2.7) and introduce the linear operator $\mathbf{T}_h^\mathrm{g} : \mathbb{T}_h \to \mathbb{F}_h$ which maps every $\mathbf{v}_h^\mathcal{T} \in \mathbb{T}_h$ onto the vector $\mathbf{v}_h^\mathcal{F} = (v_F)_{F\in\mathcal{F}_h}$ in $\mathbb{F}_h$ such that

$$\forall F \in \mathcal{F}_h^i, \qquad v_F = \begin{cases} \xi^{\mathfrak{g}_F}_{\mathbf{v}_{\mathbf{h}_h}^\mathcal{T}}(\overline{\mathbf{x}}_F) & \text{if } F \in \mathcal{F}_h^i, \\ 0 & \text{if } F \in \mathcal{F}_h^b. \end{cases} \tag{2.9}$$

The operator $\mathbf{T}_h^\mathrm{g}$ is used in a gradient reconstruction based on Green's formula and inspired from [64]. More precisely, we introduce the linear gradient operator $\mathfrak{G}_h^\mathrm{green} : \mathbb{T}_h \times \mathbb{F}_h \to [\mathbb{P}_d^0(\mathcal{T}_h)]^d$ such that, for all $(\mathbf{v}^\mathcal{T}, \mathbf{v}^\mathcal{F}) \in \mathbb{T}_h \times \mathbb{F}_h$ and all $T \in \mathcal{T}_h$,

$$\mathfrak{G}_h^\mathrm{green}(\mathbf{v}^\mathcal{T}, \mathbf{v}^\mathcal{F})|_T = \frac{1}{|T|_d} \sum_{F\in\mathcal{F}_T} |F|_{d-1}(v_F - v_T)\mathrm{n}_{T,F}. \tag{2.10}$$

The discrete space for the CCG method under examination can then be obtained as follows: (i) let $\mathcal{S}_h = \mathcal{T}_h$ and $\mathbb{V}_h = \mathbb{T}_h$; (ii) let $\mathfrak{G}_h = \mathfrak{G}_h^{\rm ccg}$ with $\mathfrak{G}_h^{\rm ccg}$ such that

$$\forall \mathbf{v}_h \in \mathbb{V}_h, \qquad \mathfrak{G}_h^{\rm ccg}(\mathbf{v}_h) = \mathfrak{G}_h^{\rm green}(\mathbf{v}_h, \mathbf{T}_h^{\rm g}(\mathbf{v}_h)). \tag{2.11}$$

The reconstruction operator defined taking $\mathfrak{G}_h = \mathfrak{G}_h^{\rm ccg}$ in (2.5) is denoted by $\mathfrak{R}_h^{\rm ccg}$, and the corresponding discrete space by $V_h^{\rm ccg} \stackrel{\rm def}{=} \mathfrak{R}_h^{\rm ccg}(\mathbb{T}_h)$. The last ingredient to formulate the discrete problem is a suitable definition of the weights in the average operator. To this end, we let, for all $F \in \mathcal{F}_h^i$,

$$\omega_{T_1,F} = \tfrac{\lambda_{T_2,F}}{\lambda_{T_1,F}+\lambda_{T_2,F}}, \qquad \omega_{T_2,F} = \tfrac{\lambda_{T_1,F}}{\lambda_{T_1,F}+\lambda_{T_2,F}},$$

where $\lambda_{T_i,F} \stackrel{\rm def}{=} \kappa|_{T_i}\mathrm{n}_F\cdot\mathrm{n}_F$ for $i \in \{1,2\}$. Set, for all $(u_h, v_h) \in V_h^{\rm ccg} \times V_h^{\rm ccg}$,

$$\begin{aligned} a_h^{\rm ccg}(u_h, v_h) \stackrel{\rm def}{=} &\int_\Omega \kappa\nabla_h u_h\cdot\nabla_h v_h - \sum_{F\in\mathcal{F}_h}\int_F [\{\kappa\nabla_h u_h\}_\omega\cdot\mathrm{n}_F[\![v_h]\!] + [\![u_h]\!]\{\kappa\nabla v_h\}_\omega\cdot\mathrm{n}_F] \\ &+ \sum_{F\in\mathcal{F}_h}\eta\frac{\gamma_F}{h_F}\int_F [\![u_h]\!][\![v_h]\!], \end{aligned} \tag{2.12}$$

with ∇_h broken gradient on $\mathcal{T}_h$, $\gamma_F = \frac{2\lambda_{T_1,F}\lambda_{T_2,F}}{\lambda_{T_1,F}+\lambda_{T_2,F}}$ on internal faces $F \subset \partial T_1 \cap \partial T_2$ and $\gamma_F = \kappa|_T\mathrm{n}_F\cdot\mathrm{n}_F$ on boundary faces $F \subset \partial T \cap \partial\Omega$. The CCG method reads

$$\text{Find } u_h \in V_h^{\rm ccg} \text{ s.t. } a_h^{\rm ccg}(u_h, v_h) = \int_\Omega f v_h \text{ for all } v_h \in V_h^{\rm ccg}. \tag{2.13}$$

The bilinear form $a_h^{\rm ccg}$ has been originally introduced by Di Pietro, Ern and Guermond [56] in the context of dG methods for degenerate advection-diffusion-reaction problems. In particular, for $\kappa = \mathbb{1}_d$, the method (2.13) coincides with the Symmetric Interior Penalty (SIP) method of Arnold [33] associated to the bilinear form

$$\begin{aligned} a_h^{\rm sip}(u_h, v_h) = &\int_\Omega \nabla_h u_h\cdot\nabla_h v_h - \sum_{F\in\mathcal{F}_h}\int_F [\{\nabla_h u_h\}\cdot\mathrm{n}_F[\![v_h]\!] + [\![u_h]\!]\{\nabla_h v_h\}\cdot\mathrm{n}_F] \\ &+ \sum_{F\in\mathcal{F}_h}\frac{\eta}{h_F}\int_F [\![u_h]\!][\![v_h]\!]. \end{aligned} \tag{2.14}$$

For further details on the link between CCG and discontinuous Galerkin methods we refer to [50, 49, 52].

2.4.3 A hybrid finite volume method

As a last example we consider the SUSHI scheme of [64]; see also [59] for a discussion on the link with the MFD methods of [42, 41]. This method is based on the gradient reconstruction (2.10), but stabilization is achieved in a rather different manner with respect to (2.12). More precisely, we define the linear residual operator $\mathfrak{r}_h : \mathbb{T}_h \times \mathbb{F}_h \to \mathbb{P}_d^0(\mathcal{P}_h)$ as follows: For all $T \in \mathcal{T}_h$ and all $F \in \mathcal{F}_T$,

$$\mathfrak{r}_h(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_{\mathcal{P}_{T,F}} = \frac{d^{\frac12}}{d_{T,F}}\left[v_F - v_T - \mathfrak{G}_h^{\rm green}(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_T\cdot(\mathbf{x}_F - \mathbf{x}_T)\right].$$

The discrete space for SUSHI method with hybrid unknowns is then obtained as follows: (i) let $\mathcal{S}_h = \mathcal{P}_h$ and $\mathbb{V}_h = \mathbb{T}_h \times \mathbb{F}_h$; (ii) let $\mathfrak{G}_h = \mathfrak{G}_h^{\rm hyb}$ with $\mathfrak{G}_h^{\rm hyb}$ such that, for all $(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}}) \in \mathbb{T}_h \times \mathbb{F}_h$, all $T \in \mathcal{T}_h$ and all $F \in \mathcal{F}_T$,

$$\mathfrak{G}_h^{\rm hyb}(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_{\mathcal{P}_{T,F}} = \mathfrak{G}_h^{\rm green}(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_T + \mathfrak{r}_h(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_{\mathcal{P}_{T,F}}\mathrm{n}_{T,F}. \tag{2.15}$$

Denote by $\mathfrak{R}_h^{\rm hyb}$ the reconstruction operator defined by (2.5) with $\mathfrak{G}_h = \mathfrak{G}_h^{\rm hyb}$. The SUSHI method with hybrid unknowns reads

$$\text{Find } u_h \in V_h^{\rm hyb} \text{ s.t. } a_h^{\rm sushi}(u_h, v_h) = \int_\Omega f v_h \text{ for all } v_h \in V_h^{\rm hyb},$$

with $a_h^{\rm sushi}(u_h, v_h) \stackrel{\rm def}{=} \int_\Omega \kappa \nabla_h u_h \cdot \nabla_h v_h$ and ∇_h broken gradient on $\mathcal{P}_h$. Alternatively, one can obtain a cell centered version by setting $\mathbb{V}_h = \mathbb{T}_h$ and replacing $\mathfrak{G}_h^{\rm hyb}$ defined by (2.15) by $\mathfrak{G}_h = \mathfrak{G}_h^{\rm cc}$ with $\mathfrak{G}_h^{\rm cc}$ such that

$$\forall \mathbf{v}_h \in \mathbb{T}_h, \qquad \mathfrak{G}_h^{\rm cc}(\mathbf{v}_h) = \mathfrak{G}_h^{\rm hyb}(\mathbf{v}_h, \mathbf{T}_h^{\rm g}(\mathbf{v}_h)), \tag{2.16}$$

with $\mathbf{T}_h^{\rm g}$ defined by (2.9). This variant coincides with the version proposed in [64] for homogeneous κ, but it has the advantage to reproduce piecewise affine solution to (2.4) on $\mathcal{T}_h$ when κ is heterogeneous. The resulting discrete spaces is labeled $V_h^{\rm cc}$ in Table 3.4.

This space allows a **Flux Reconstruction Operator**:
$\mathfrak{F}_h(\mathbf{v}_h^{\mathcal{T}}, \mathbf{v}_h^{\mathcal{F}})|_{F,T} = \sum_{F' \in \mathcal{F}_T} A_T^{FF'}(v_T - v_{F'})$,
where:
$A_T^{FF'} = \sum_{F" \in \mathcal{F}_T} y^{F"F} . \kappa_{T,F''} y^{F"F}$
$y^{F,F} = \frac{|F|}{|T| \mathrm{n}_{T,F}} + \frac{\sqrt{d}}{d_{T,F}}(1 - \frac{|F|}{|T|} \mathrm{n}_{T,F} \cdot (\mathbf{x}_F - \mathbf{x}_T)) \mathrm{n}_{T,F}$
$y^{F,F'} = \frac{|F'|}{|T| \mathrm{n}_{T,F'}} - \frac{\sqrt{d}}{d_{T,F}|T|} |F'| \mathrm{n}_{T,F'} \cdot (\mathbf{x}_F - \mathbf{x}_T) \mathrm{n}_{T,F}$

2.4.4 Integration

To compute integral expression with the different methods, considering $\mathbb{V}_h$ a functional space, $v_h \in \mathbb{V}_h$, for $T \in \mathcal{T}_h$, for each $S \in \mathcal{S}_h$ and $S \subset T$ for all $x \in S$, as $\mathfrak{G}_h(\mathbf{v}_h)(x)$ is constant and $v_h(x)$ is affine, we approximate $\int_T v_h(x)dx \approx |T| v(\mathbf{x}_T)$ and $\int_T \nabla f(x)dx \approx \sum_{S \subset T} |S| \nabla_h f|_S$.

For $u \in \mathbb{V}$, $u_h \in V_h$ with $u_T = f(\mathbf{x}_T)$ we can write then $\int_\Omega f(x)dx \approx \sum_{T \in \mathcal{T}_h} |T| u_T$ and $\int_\Omega \nabla f(x)dx \approx \sum_{T \in \mathcal{T}_h} \sum_{S \subset T} |T| \nabla_h u|_S$

2.5 Extensions for multiscale methods

In this section, we extend the formalism presented in the previous section for multiscale methods. These methods have been introduced to solve PDE systems modeling phenomena that are governed by physical processes occurring on a wide range of time and/or length scales. This kind of problems cannot be solved on the finest grid due to time and memory limitations. They consist in incorporating fine-scale information into a set of coarse-scale equations in a way that is consistent with the local properties of the mathematical model on the unresolved subscale(s). In geoscience, multiscale methods are considered for the simulation of pressure and (phase) velocities in porous media flow. Multiscale behavior in porous media flow are due to heterogeneities in rock and sand formations which are reflected in the permeability tensor used in the governing partial differential equations. To accurately resolve the pressure distribution, it is necessary to account for the influence of fine-scale variations in the coefficients of the permeability tensor. For incompressible flow, the pressure equation reduces to the following variable coefficient Poisson equation:

$$\begin{aligned} v &= -\kappa \nabla p \text{ on } \Omega, \\ \nabla \cdot v &= q \text{ on } \Omega, \\ p &= g \text{ on } \partial\Omega_D, \\ \partial_n p &= 0 \text{ on } \partial\Omega_N = \partial\Omega \setminus \partial\Omega_D, \end{aligned} \tag{2.17}$$

where

- q is a source term,

- κ stands for the symmetric positive definite permeability tensor that typically has a multiscale structure due to the strongly heterogeneous nature of natural porous media,
- $\partial\Omega_D$ (respectively $\partial\Omega_N$) is a subset of the boundary $\partial\Omega$ of Ω where Dirichlet (respectively Neumann) boundary conditions are enforced.

In [91] an interesting overview on multiscale methods is done by Kippe V., Aarnes J. E. and Lie K. A. They describe various methods and in particular the multiscale finite-element method (MsFEM) [73], the variational multiscale method [76], the mixed multiscale finite-element method (MxMsFEM) [44], and the multiscale finite-volume method (MsFVM)[75]. We present these methods with more details in Annexe C where we present the main ideas of their work. These methods are all based on lowest order methods and, except for the multiscale finite volume method, they are formulated as Petrov-Galerkin schemes. All these elements make it possible to formulate them within our mathematical formalism. We have therefore extend our unified mathematical formalism to describe such methods. In this section we present this extension applied in particular to a variation of the MxMsFEM method described in §C.3 of Annexe C. This variation consists in using the SUSHI method presented in §2.4.3 as lowest order method, and in formulating the coarse level method with the discontinuous Galerkin formalism to deal with the discontinuity of the basis functions at interfaces in the same spirit as for the the CCG method of §2.4.2.

2.5.1 Multiscale ingredients

Like the previous methods presented in this chapter, multiscale methods are based on a multiscale functional space built from the following ingredients:

- a mesh partitioning the domain Ω;
- a space of degree of freedoms $\mathbb{V}_h$;
- a gradient and a reconstruction operator that enable us to map vectors of $\mathbb{V}_h$ to functions.

The main differences rely in the introduction of a two level mesh and basis functions allowing to handle a gradient reconstruction operators piecewise constant on the thin elements of $\mathcal{T}_h$, whereas in the methods presented in Chapter 2 only piecewise constant operators on the element of $\mathcal{S}_h$ submesh of $\mathcal{T}_h$ are considered.

2.5.2 Mesh settings

Multiscale methods are based on a coarse and a fine grid. For the following section we introduce a few notations: we denote $\mathcal{T}_h$ and $\mathcal{T}_H$ respectively the fine and the coarse grid. The exponent ^{c} and ^{f} are used for respectively elements related to the coarse grid and the fine grid. We denote τ^c and σ^c respectively cell and face elements of $\mathcal{T}_H$ and τ^f and σ^f, cell and face elements of $\mathcal{T}_h$. Let $\mathcal{T}_h$ be a discretization of Ω and $\mathcal{T}_H$ be a coarse mesh obtained by a coarsening of $\mathcal{T}_h$. Let $\mathcal{F}_h$ be the set of faces of $\mathcal{T}_h$ and $\mathcal{F}_H$, the set of faces of $\mathcal{T}_H$. We introduce the following notations:

- $\forall \tau^c \in \mathcal{T}_H$, we define $\mathcal{G}_{\tau^c} \stackrel{\text{def}}{=} (\bigcup \tau)_{\tau\in\mathcal{T}_h, \tau\cap\tau^c\neq\emptyset}$ as the set of fine elements of the coarse element τ^c and $\mathcal{F}^c_{\tau^c} \stackrel{\text{def}}{=} \{\sigma^c \in \mathcal{F}_H \mid F \subset \partial\tau^c\}$.
- $\forall \sigma^c \in \mathcal{F}_H$, we define $\mathcal{G}_{\sigma^c} \stackrel{\text{def}}{=} (\bigcup \sigma)_{\sigma\in\mathcal{F}_h, \sigma\cap\sigma^c\neq\emptyset}$.

We say that $(\mathcal{T}_h, \mathcal{T}_H)$ is a two level mesh partitioning Ω.

If σ is shared by two cells τ_1 and τ_2, we denote τ_σ the union $\tau_1 \cup \tau_2$. σ can be denoted also $\tau_1 \cap \tau_2$ or $\tau_{1,2}$.

We denote $\mathcal{F}_H^b \overset{\text{def}}{=} \{\sigma^c \in \mathcal{F}_H \mid \sigma^c \cap \partial\Omega \neq \emptyset\}$ and $\mathcal{F}_H^i \overset{\text{def}}{=} \{\sigma^c \in \mathcal{F}_H \mid \sigma^c \cap \partial\Omega = \emptyset\}$

We define $\Gamma_D \overset{\text{def}}{=} \{\sigma^c \in \mathcal{F}_H \mid \sigma^c \cap \partial\Omega_D \neq \emptyset\}$ and $\Gamma_N \overset{\text{def}}{=} \{\sigma^c \in Fh^c \mid \sigma^c \cap \partial\Omega_N \neq \emptyset\}$.

2.5.3 The Hybrid Multiscale Method

The Hybrid Multiscale method (HMsM) is a variation of the MxMsFEM presented in Annexe C.3. The idea is to use the SUSHI method instead of the standard FE methods to build the basis functions and to design the coarse level formulation. The main principles remain to compute with a lowest order method defined on the coarse grid $\mathcal{T}_H$ a solution from which we interpolate the solution on the fine grid $\mathcal{T}_h^f$.

Basis functions For each $\sigma^c = \tau_1^c \cap \tau_2^c \in \mathcal{F}_H^i \cup \Gamma_D$, ($\sigma^c$, τ_1^c and τ_2^c are represented by Fc, K an L in figure 2.3) we define ϕ_{σ_c} basis functions as follows:

- If $\sigma^c \in (\mathcal{F}_H^i)$ and $\sigma^c = \tau_1^c \cap \tau_2^c$, $\Omega_{\sigma_c} \overset{\text{def}}{=} \text{supp}(\phi_{\sigma^c})$ is defined by $\mathcal{G}_{\tau_1^c} \cup \mathcal{G}_{\tau_2^c}$ and ϕ_{σ_c} is solution of

$$\begin{aligned} -\nabla \cdot (\kappa \nabla \phi_{\sigma^c}) &= w_1 \text{ on } \tau_1^c, \\ -\nabla \cdot (\kappa \nabla \phi_{\sigma^c}) &= -w_2 \text{ on} \tau_2^c, \\ \kappa \nabla \phi_{\sigma^c} \cdot \mathrm{n} &= 0 \text{ on } \partial\Omega_{\sigma^c} \end{aligned} \tag{2.18}$$

  In (2.18), the functions $w_i, i \in \{1, 2\}$ are weight functions defined by

$$w_i = \frac{\text{trace}(\kappa)}{\int_{\tau_i} \text{trace}(\kappa)} \text{ if } q|_{\tau_i} = 0, w_i = q \text{ otherwise} \tag{2.19}$$

  where trace(κ) is the trace of the permeability tensor κ.

  The basis functions are computed up to one constant. Here we assumed the unit normal vector n_{σ^c} is oriented outward with respect to τ_1^c.

- If $\sigma^c \in \Gamma_D \cap \mathcal{F}_{\tau^c}^c$, $\Omega_{\sigma_c} \overset{\text{def}}{=} \text{supp}(\phi_{\sigma^c})$ is defined by $\mathcal{G}_{\tau^c}$ and ϕ_{σ_c} is solution of:

$$\begin{cases} -\nabla \cdot (\kappa \nabla \phi_{\sigma^c}) = w \text{ on } \tau^c, \\ \kappa \nabla \phi_{\sigma^c} \cdot \mathrm{n} = 0 \text{ on } \partial\tau^c \setminus \sigma^c, \\ -\kappa \nabla \phi_{\sigma^c} \cdot \mathrm{n}_{\sigma^c} = \frac{\kappa \mathrm{n}_{\sigma^c}}{\int_{\sigma^c} \kappa \mathrm{n}} \text{ on } \sigma^c. \end{cases} \tag{2.20}$$

  The function w is also given by (C.11). As previously, we assumed that n_{σ^c} is oriented outwards with respect to τ^c.

By defining the basis functions in that way, we can notice that:

$$\int_{\partial\tau_1^c} -\kappa \nabla \phi_{\sigma^c} \cdot \mathrm{n} = -\int_{\partial\tau_2^c} -\kappa \nabla \phi_{\sigma^c} \cdot \mathrm{n} = \int_{\sigma^c} -\kappa \nabla \phi_{\sigma^c} \cdot \mathrm{n} = 1.$$

The functions ϕ_{σ^c} are thus defined to set a unit flux on σ^c and no fluxes on $\tau_{1,2}^c$. The term trace(κ) in the weight functions or in the boundary conditions of (2.18) and (2.20) enables to weight the fine fluxes at the fine scale according to the permeability field.

To solve the local problems (2.18) and (2.20) we use the SUSHI method. Let $U_{\Omega_{\sigma_c}}^{sushi}$, an SUSHI space defined on Ω_{σ_c}, the variational formulation reads: find $u_h \in U_{\Omega_{\sigma_c}}^{sushi}$ so that:
$\forall v_h \in U_{\Omega_{\sigma_c}}^{sushi}, a_h(u_h, v_h) = b_h(v_h)$ where

$$\begin{cases} a_h(u_h, v_h) & \overset{\text{def}}{=} \int_{\Omega_{\sigma_c}} \kappa \nabla_h u_h \cdot \nabla_h v_h \\ b_h(v_h) & \overset{\text{def}}{=} \int_{\Omega_{\sigma_c}} w v_h \end{cases}$$

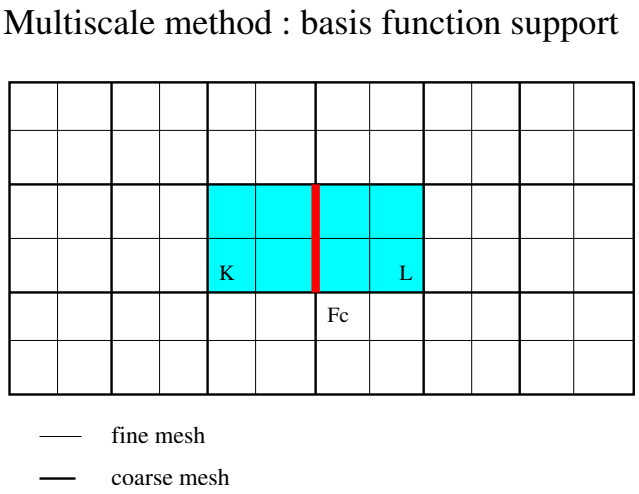


Figure 2.3: Basis function

Hybrid multiscale functional space

The Hybrid Multiscale method consists in finding discrete solutions of (2.17) expressed as linear combination of the multiscale basis functions ϕ and the characteristic function of coarse element i.e.:

$$\mathbf{u}_h = \sum_{\tau^c \in \mathcal{T}_H} u_{\tau_c} \chi_{\tau_c} + \sum_{\sigma_c \in \mathcal{F}_H} v_{\sigma_c} \phi_{\sigma_c}$$

where $(u_{\tau_c}, v_{\sigma_c}) \in \mathbb{R}^{\mathcal{T}_H \times \mathcal{F}_H}$, χ_{τ_c} is the characteristic function of $\tau_c \in \mathcal{T}_H$.

Let $\mathbb{T}_H = \mathbb{R}^{\mathcal{T}_H}$ and $\mathbb{F}_H = \mathbb{R}^{\mathcal{F}_H}$, in the unified framework formalism, it is equivalent to set $\mathbb{V}_H = \mathbb{T}_H \times \mathbb{F}_H$ with DOFs on both coarse cells and coarse faces and to define the operator $\mathfrak{R}_H^{hms}$ mapping every vector of DOFs $(u_H, v_H) \in \mathbb{V}_H$ onto the function $\mathfrak{R}_H(u_H, v_H)^{hms}$ defined $\forall \tau_c \in \mathcal{T}_H, \forall \mathbf{x} \in \tau^c$ as follows:

$$\mathbf{u}|_{\tau_c}(\mathbf{x}) = u_{\tau_c} + \sum_{\sigma_c \in \mathcal{F}_{\tau_c}^c} v_{\sigma_c} \phi_{\sigma_c}(\mathbf{x})$$

We define a discrete functional space for hybrid multiscale method as $V^{hms} \overset{\text{def}}{=} \mathfrak{R}_H^{hms}(\mathbb{T}_H, \mathbb{F}_H)$

The concept of gradient reconstruction operator $\mathfrak{G}_h$ defined in Chapter 2 can be extended by setting:

- $\mathcal{S}_h = \mathcal{T}_H$
- $\forall \mathbf{v}_h \in \mathbb{V}_h, \forall \tau_c \in \mathcal{T}_H, \mathfrak{G}_h^{\text{hms}}(\mathbf{v}_h)$ is defined as:

$$\mathfrak{G}_h^{\text{hms}}(\mathbf{v}_h) = \sum_{\sigma_c \in \mathcal{F}_{\tau_c}^c} v_{\sigma_c} \nabla \phi_{\sigma_c}$$

Here, $\mathfrak{G}_h$ does not define elements of $\mathbb{P}_d^0(\mathcal{S}_h)$ but rather a function that is piecewise constant on the fine elements of $\mathcal{S}_h$.

The coarse level variational formulation

Unlike for the original MxMsFEM method, we have to deal with the discontinuity of our basis functions on each coarse element. To this end, we rely on a weak enforcement of continuity across

interfaces inspired by the SIP method of Arnold [33]. The variational formulation for the coarse problem reads:

Find $u_h \in U^{hms}$, so that $\forall v_h \in U^{hms}, a_H^{hms}(u_h, v_h) = b_H(v_h)$ where

$$\begin{aligned} a_H^{hms}(u_h, v_h) \stackrel{\text{def}}{=} & \int_\Omega \kappa \nabla_H u_h \cdot \nabla_H v_h \\ & - \sum_{\sigma^c \in \mathcal{F}_H} \int_{\sigma^c} (\llbracket u_h \rrbracket (\{\kappa \nabla_H v_h\} \cdot \mathrm{n}_{\sigma^c}) + (\{\kappa \nabla_H u_h\} \cdot \mathrm{n}_{\sigma^c}) \llbracket v_h \rrbracket) \\ & + \sum_{\sigma^c \in \mathcal{F}_H} \int_{\sigma^c} \frac{\eta}{h} \llbracket u_h \rrbracket \llbracket v_h \rrbracket \end{aligned} \tag{2.21}$$

where we use a weighted average operator to deal with the heterogeneity of κ.

The source term q of the model problem (2.17) is embedded in the construction of the basis functions, more exactly in the computation of the source term w involved in the local PDE problem (2.5.3) defining a basis function ϕ_{σ^c}. Indeed this term depends on q when $q(x) \neq 0$ on Ω_{σ^c}. For this reason we have generally $b_H(v_h) = 0$.

The right hand side of the linear system is filled while evaluating the bilinear form $a_H^{hms}(u_h, v_h)$, we deal with the dirichlet boundary conditions. This is done at the evaluation of the terms corresponding the weak enformcement of continuity across the interfaces $\sigma^c \in \Gamma_D$.

2.5.4 Hybrid Multiscale algorithm

The hybrid multiscale method defines a two steps algorithm: the first step consists in building a coarse linear system with the coarse level method defined previously to compute a coarse solution. The second step consists in downscaling that solution on the fine level.

Coarse linear system Assembly

The coarse linear system is built by the evaluation of the bilinear forms (2.21) which is composed of the two main terms:

$$\int_\tau \kappa \nabla_H u_h \cdot \nabla_H v_h \text{ and } \int_\sigma (\llbracket u_h \rrbracket (\{\kappa \nabla_H v_h\} \cdot \mathrm{n}_\sigma)$$

Writting $u_h = \sum_{\tau \in \mathcal{T}_H} u_\tau \chi_\tau + \sum_{\sigma \in \mathcal{F}_H} v_\sigma \phi_\sigma$, we can notice that:

- $$\begin{aligned} \int_\Omega \kappa \nabla_H u_h \cdot \nabla_H v_h &= \sum_{\tau \in \mathcal{T}_H} \int_{\tau^c} \kappa \nabla_H u_h \cdot \nabla_H v_h \\ &= \sum_{\tau^c \in \mathcal{T}_H} \sum_{\sigma_1, \sigma_2 \in \mathcal{F}_{\tau^c}^c} v_{\sigma_1} v_{\sigma_2} \int_{\tau^c} \kappa \nabla_H \phi_{\sigma_1} \cdot \nabla_H \phi_{\sigma_2} \end{aligned} \tag{2.22}$$

- integrating $\int_\sigma (\llbracket u_h \rrbracket (\{\kappa \nabla_H v_h\} \cdot \mathrm{n}_\sigma)$ with one quadrature point on the barycenter $\mathbf{x}_\sigma$, we have for $\sigma = \tau_1 \cap \tau_2$:

$$\begin{aligned} \int_\sigma (\llbracket u_h \rrbracket (\{\kappa \nabla_H u_h\} \cdot \mathrm{n}_\sigma) &= \int_\sigma (u_h|_{\tau_1} - u_h|_{\tau_2})(\kappa \nabla_H u_h \cdot \mathrm{n}_\sigma) \\ &= |\sigma| (u_h|_{\tau_1}(\mathbf{x}_\sigma) - u_h|_{\tau_2}(\mathbf{x}_\sigma))(\kappa \nabla_H u_h \cdot \mathrm{n}_\sigma)(\mathbf{x}_\sigma) \\ &= |\sigma| (\frac{1}{|\sigma|} \int_\sigma u_h|_{\tau_1} - \frac{1}{|\sigma|} \int_\sigma u_h|_{\tau_2}) \mathbf{v}_\sigma \frac{1}{|\sigma|} \int_\sigma \kappa \nabla_H \phi_\sigma \cdot \mathrm{n}_\sigma \end{aligned} \tag{2.23}$$

Noting that $\int_\sigma \kappa \nabla_H \phi_\sigma \cdot \mathrm{n}_\sigma = 1$, we obtain finally an assembly phase based, for $\tau \in \mathcal{T}_H$, $\sigma_1 \in \mathcal{F}_\tau$ and $\sigma_2 \in \mathcal{F}_\tau$, on the evaluation of the terms $\int_{\tau_c} \kappa \nabla_H \phi_{\sigma_1} \cdot \nabla_H \phi_{\sigma_2}$ and $\frac{1}{|\sigma_1|} \int_{\sigma_1} \phi_{\sigma_2}$

Algebraic considerations

The mathematical functions and basis functions involved in the multiscale method have all algebraic representations:

- Elements $u_h \in V^{hms}$ are represented by vectors $(\mathbf{u}_H, \mathbf{v}_H) \in \mathbb{T}_H \times \mathbb{F}_H$ at the coarse level and $(\mathbf{u}_h, \mathbf{v}_h) \in \mathbb{T}_h \times \mathbb{F}_h$ at the fine level;
- the basis functions ϕ_{σ^c} are represented by vectors $(\mathbf{fu}_{\sigma^c}, \mathbf{fv}_{\sigma^c}) \in \mathbb{T}_h \times \mathbb{F}_h$ and $\nabla\phi_{\sigma^c}$ by $\mathbf{g}_{\sigma^c} \in \mathbb{F}_h$. The component values of these vectors are null for all mesh elements indices τ^f and σ^f except those belonging to Ω_{σ^c}.

The computation of the basis functions consists in solving a collection of independent local PDE problems which leads to build and solve a collection of independent local linear systems. All these computations are independent and can be done in parallel.

Fine solution reconstruction

The fine solution p^f and $v^f = -\kappa\nabla p^f$ is recovered from the coarse solution $(\mathbf{u}_H, \mathbf{v}_H) \in \mathbb{T}_H \times \mathbb{F}_H$, with $\mathbf{u}_H = (u_{\tau^c})$ and $\mathbf{u}_H = (v_{\sigma^c})$, writting:

$p^f = \sum_{\tau^c \in \mathcal{T}_H} u_{\tau_c}\chi_{\tau^c} + \sum_{\sigma^c \in \mathcal{F}_H} v_{\sigma^c}\phi_{\sigma^c}$ and $v^f = \mathfrak{F}_h(u^c, v^c) = \sum_{\sigma^c \in \mathcal{F}_H} v_{\sigma^c}\mathfrak{F}_h(\phi_{\sigma^c})$, where $\mathfrak{F}_h$ is the flux reconstruction operator introduced in §2.4.3.

Let $(\mathbf{up}_h, \mathbf{vp}_h) \in \mathbb{T}_h \times \mathbb{F}_h$ be the algebraic representation of p^f and $\mathbf{vv}_h$ the algebraic representation of v^f. Let $(\mathbf{uf}_{\sigma^c}, \mathbf{vf}_{\sigma^c}) \in \mathbb{T}_h \times \mathbb{F}_h$ and $\mathbf{g}_{\sigma^c} \in \mathbb{F}_h$ be the algebraic representation of respectively ϕ_{σ^c} and $\mathfrak{F}_h(\phi_{\sigma^c})$. The computation of $(\mathbf{up}_h, \mathbf{vp}_h)$ consists in iterating on each coarse cells $\tau^c \in \mathcal{T}_H$ then in updating $(\mathbf{up}_h, \mathbf{vp}_h)$ and $\mathbf{vv}_h$ evaluating the following vector linear combinations:

$$\mathbf{up}_h += u_{\tau^c} + \sum_{\sigma^c \in \mathcal{F}^c_{\tau^c}} v_{\sigma^c}\mathbf{uf}_{\sigma^c}$$

$$\mathbf{vp}_h += \sum_{\sigma^c \in \mathcal{F}^c_{\tau^c}} v_{\sigma^c}\mathbf{vf}_{\sigma^c}$$

$$\mathbf{vv}_h += \sum_{\sigma^c \in \mathcal{F}^c_{\tau^c}} v_{\sigma^c}\mathbf{g}_{\sigma^c}$$

Chapter 3

Computational setting

This chapter is partly taken from our Bit Numerical Mathematics [48] up to section 2.4, then in the last section we turn to the mathematical extension to multiscale problems.

DS(E)Ls are an established means to break the complexity of applications by allowing each contributor to express themselves in a language as close as possible to their technical jargon. In the context of scientific computing, complexity spans different levels:

(i) *Modeling.* Modelers investigate more and more comprehensive physical models expressed in terms of (systems of) Partial Differential Equations (PDEs) possibly completed by algebraic closure laws;

(ii) *Discretization.* Numericians confront with an increasing number of discretization methods which are potentially suited to convert the PDE problem into a system of algebraic equations. Disposing of different discretization methods within a unified framework is highly beneficial since it allows to identify the most efficient choice for the problem at hand;

(iii) *Solution.* Several low-level numerical packages are available to solve systems of algebraic equations. Their performance in terms of computational efficiency and stability is strongly related to both the features of the matrix to solve (symmetry, fill-in pattern, etc.) and the underlying hardware architecture;

(iv) *Software design.* Finally, computer scientists design low-level data structures and algorithms that benefit from the evolution of both hardware architectures and languages to ensure the overall efficiency.

Ideally, a software platform should allow contributors at each level to focus on a specific aspect of the problem without being hindered by the interaction with the other levels.

The mathematical framework presented in Chapter 2 enables to describe a large family of lowest order methods in a unified Galerkin formalism. As for FE methods, this formalism gives the opportunity to design of a high level language aimed at numericians that allows to express lowest-order methods using a syntax as close as possible to the mathematical notation.

For our diffusion model problem (2.4), such a DSEL will for instance achieve to express the variational discretization formulation 2.13 with the programming counterpart in listing 3.1.

Listing 3.1: Diffusion problem implementation

```
MeshType Th;
Real K;
auto Vh = new CGGSpace(Th);
auto u = *Vh->trial();
```

```
auto v = *Vh->test();
auto lambda = eta*gamma/H();
BilinearForm a =
  integrate(allCells(Th),dot(K*grad(u),grad(v))) +
  integrate(allFaces(Th),jump(u)*dot(N(),avg(grad(v))) -
                                    dot(N(),avg(K*grad(u)))*jump(v) +
                                    lambda*jump(u)*jump(v);
LinearForm b =
  integrate(allCells(Th),f*v);
```

The **Key ingredients** to design a DSEL are:

1. Meta-programming techniques that consist in writing programs that transform types at compile time
2. Generic programming techniques that consist in designing generic components composed of abstract programs with generic types
3. Generative programming techniques that consist in generating concrete programs, transforming types with meta-programs to create concrete types to use with abstract programs of generic components
4. Expression template techniques [28, 34, 92] that consist in representing problems with expression trees and using tools to describe, parse and evaluate theses trees.

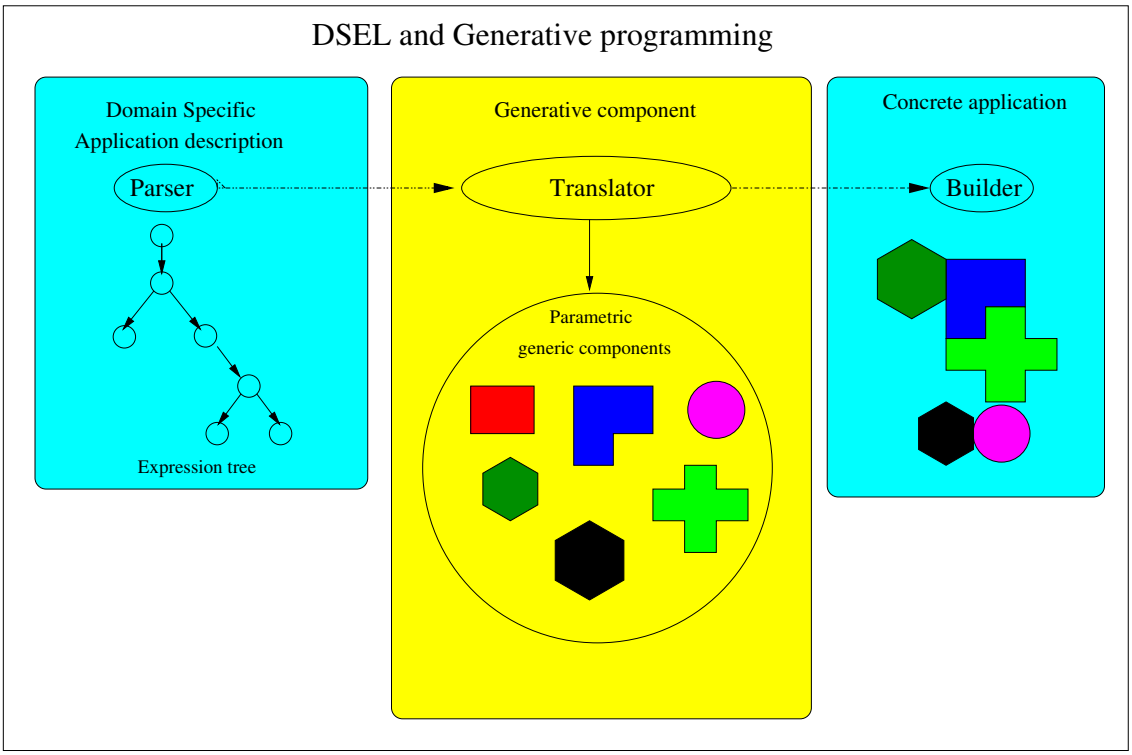


Figure 3.1: Generative programming

Applying all these techniques, it is possible to represent a problem with an expression tree. Parsing this tree at compile time, using meta programming tools to introspect the expression, it is possible to select generic components, to link them together to assemble and generate a concrete program. The execution of this program consists in evaluating the tree at runtime, executing the

concrete instance of the selected components to build a linear system, solve it to find the solution of the problem. Figure 3.1 illustrates the main principles of the generative programming workflow: on the left we have the description of the problem with a tree expression, in the middle the generative process at compile time, and on the right the execution process at runtime.

Using these principles, we have designed a DSEL that enables us to express and define linear and bilinear forms. The terminals of our language are composed of symbols representing C++ objects with base types (Real or Integer) and with types representing discrete variables, functions, test and trial functions. Our language uses the standard C++ binary operators (+,-,*,/), the binary operator **dot**`(.,.)` representing the scalar product of vector expressions, unary operators representing standard differential operators like **grad**`(.)` and **div**`(.)`. The language is completed by a number a specific keywords **integrate**`(.,.)`, **N**`()` and **H**`()`.

The **integrate**`(.,.)` keyword associates a collection of mesh entities to linear and bilinear expressions.

N`()` and **H**`()` are free functions returning discrete variables containing respectively the precomputed values of n_F and h_F of the mesh faces of $\mathcal{T}_h$.

Figure 3.2 illustrates the link between the front end of a language composed of high level user structures and keywords and the back-end composed of low level algebraic structures and generic algorithms.

The generative programming is nowadays a well established technology. The technics of expression template and meta-programming in `C++` have been developed since the middle of the 90s [28, 34, 92]. They have been already applied in scientific computing in projects like `blitz` in the early 2000s. Since then, they have been applied intensively in general purpose libraries like Spirit[15], Phoenix[13]. The `Boost.Proto` library of Niebler [83] is a framework of this kind that provides user friendly tools to design DSLs in `C++`. It is somehow a DSEL to design user DSELs. This library, part of the Boost project, has been successfully used in projects like Phoenix, NT2[12] and is nowadays very mature. We have based our development on this framework for all the useful tools its provides to design languages, to parse and introspect expressions and to generate algorithms with low level structures.

The material in this chapter is largely taken from [48] and [68] and is organized as follows:

In §3.1 we present the algebraic back-end for the DSEL. In particular, we introduce the programming counterpart of meshes and spaces of DOFs.

In §3.2 we present the functional front-end of the DSEL, which is based on the concept of function space. We focus on the originality of these concepts which are inspired by the `Feel++` library, but are here extended to account for the peculiarities of the methods at hand.

In §3.3 we present a DSEL framework that allows to build complex expressions representing bilinear and linear forms. We show how different evaluation contexts enable us to generate algorithms to solve the discrete problem by exploiting the linear algebra layer. We present also a few extensions that handle:

- vector expressions representing 2D and 3D fields like for instance the velocity fields or properties of collection of entities;
- general boundary conditions with extra constraints or equations that modify the linear system to solve.

In §3.4 we present the extension of computational framwork to multiscale methods.

Finally in §3.5 we provide several numerical examples to assess the performance of the proposed approach. A special care is devoted to the evaluation of the overhead of the DSEL and to the comparison with more standard methods/implementations.

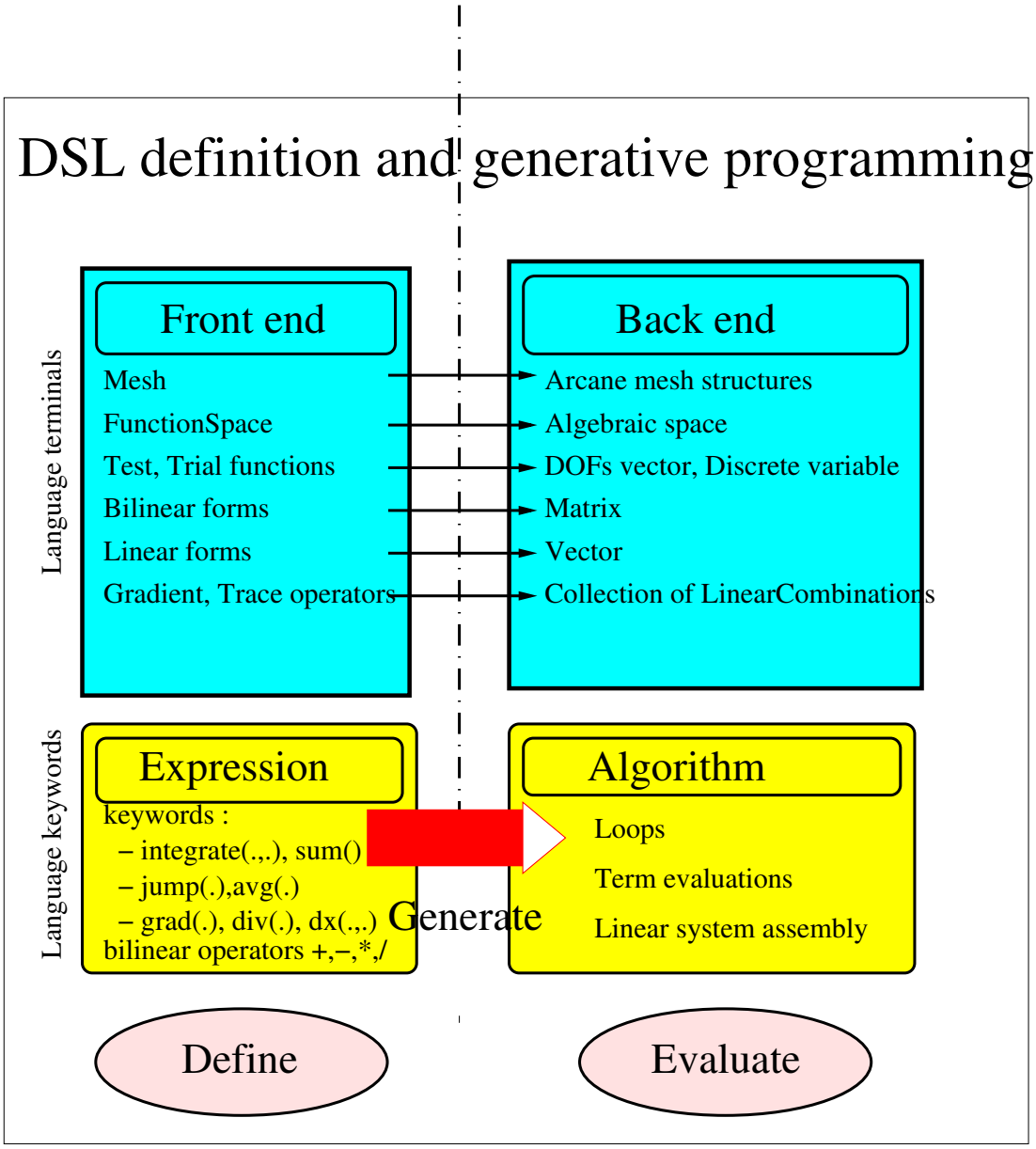


Figure 3.2: DSEL and generative programming

3.1 Algebraic back-end

In this section we focus on the elementary ingredients used to build the terms appearing in the linear and bilinear forms of Chapter 2, which constitute the back-end of the DSEL presented in §3.3.

3.1.1 Mesh

The mesh concept is an important ingredient of the mathematical frame. Mesh types and data structures are a very standard issue and different kinds of implementation already exists in various frameworks. We have developed above Arcane mesh data structures a **mesh concept** defining (i) `MeshType::dim` the space dimension, (ii) the subtypes `Cell`, `Face` and `Node` for a mesh element of dimension respectively `MeshType::dim`, `MeshType::dim-1` and 0. The free functions listed in table 3.1 are provided to manipulate the mesh, to extract different parts, and to access to the mesh connectivity. Observe that the notion of geometric element type is absent from the mesh concept, as it is irrelevant for the methods considered in this work.

Table 3.1: Connectivity and mesh element extraction tools. $\mathcal{P}_h^{key}$ is a subset of $\mathcal{T}_h$ or $\mathcal{F}_h$ associated to a key label `<key>`

`All<Item>::items(<mesh>)`	elements of the mesh, $\mathcal{T}_h$ for cells, $\mathcal{F}_h$ for faces
`Boundary<Item>::items(<mesh>)`	elements of the boundary of the mesh $\mathcal{T}_h^b$ for cells, $\mathcal{F}_h^b$ for faces
`Internal<Item>::items(<mesh>)`	$\mathcal{T}_h^i$ for cells, $\mathcal{F}_h^b$ for faces
`All<Item>::items(<mesh>,<key>)`	$\mathcal{P}_h^{key}$
`Boundary<Item>::items(<mesh>,<key>)`	$\mathcal{P}_h^{key} \cap \mathcal{T}_h^b$ for cells, $\mathcal{P}_h^{key} \cap \mathcal{F}_h^b$ for faces
`Internal<Item>::items(<mesh>,<key>)`	$\mathcal{P}_h^{key} \cap \mathcal{T}_h^i$ for cells, $\mathcal{P}_h^{key} \cap \mathcal{F}_h^i$ for faces
`elements<Item>::items(<mesh>,<item>)`	the set of elements of <mesh> connected to <item>,$\mathcal{T}_F$ or $\mathcal{F}_T$

A user friendly version of these free functions are listed in Table 3.2

Table 3.2: Mesh accessors for an object `Th` of type `Mesh`

Item set	Accessor
$\mathcal{T}_h$	`allCells(Th)`
$\mathcal{F}_h$	`allFaces(Th)`
$\mathcal{F}_h^i$	`interfaces(Th)`
$\mathcal{F}_h^b$	`boundaryFaces(Th)`

SubMesh, submesh element identification. Tags (listing 3.2) have been created to identify: (i) the back or the front position of a cell regarding a face; (ii) the two submesh types identity and pyramidal.

Listing 3.2: Tags definition for submesh identification

```
namespace mesh {
  //!enum type defining cell position with respect to face
  typedef enum {eBack,eFront} eCellFacePosType;
  namespace tag {
    struct back{};        //! back position tag
```

```
    struct front{};       //! front position tag
    struct boundary{};    //! boundary position tag
    namespace submesh {
      struct Th {};       //! idendity submesh tag
      struct Ph {};       //! Pyramidal submesh tag
    };
  };
};
```

The identity submesh $\mathcal{S}_h = \mathcal{T}_h$ is represented by `mesh::tag::submesh::Th`.

The pyramidal submesh $\mathcal{S}_h$ of $\mathcal{T}_h$ is represented by `mesh::tag::submesh::Ph`, and each element $S \in \mathcal{S}_h$ is uniquely identified by a face element $F \in \mathcal{F}_h$ and a position $p \in \{back, front\}$ relatively to F or by a cell element $T \in \mathcal{T}_h$ and an integer relative position of a face $F \in \mathcal{F}_T$.

3.1.2 Vector spaces, degrees of freedom and discrete variables

The `class Variable` with template parameters `ItemT` and `ValueT` manages vectors of values of type `ValueT` and provides data accessors to these values with either mesh elements of type `ItemT`, integer ids or iterators identifying these elements. Instances of the class `Variable` are managed by `VariableMng`, a class that associates each variable to its unique string key label corresponding to the variable name.

We represent vector spaces of DOFs $\mathbb{V}_h$ by instances of `VariableMng`, and vector space elements $\mathbf{v}_h \in \mathbb{V}_h$, identified by their name by instances of `Variable`. The vector of DOF values is represented by the variable vector values and the different data accessors that enable us to map mesh entities to DOF values.

3.1.3 Assembly

The point of view presented in Chapter 2 naturally leads to finite element-like assembly of local contributions stemming from integrals over elements or faces. However, a few major differences have to be taken into account: (i) the stencil of the local contributions may vary from term to term; (ii) the stencil may be data-dependent, as is the case for the methods of Chapter 2 based on the L-construction; (iii) the stencil may be non-local, as DOFs from neighboring elements may be used in local reconstructions. All of the above facts invalidate the classical approach based on a global table of DOFs inferred from a mesh and a finite element (in the sense of Ciarlet [47, pag. 93]).

Linear combination Our approach to meet the above requirements is to (i) drop the concept of local element, and to refer to DOFs by a unique global index; (ii) introduce the concept of linear combination, which realizes a linear application from $\mathbb{V}_h$ onto the space $\mathbb{T}_r$ of real tensors of order $r \leq 2$.

In practice, the main ingredient of linear combination is an efficient mapping of the DOFs in $\mathbb{V}_h$ onto the corresponding coefficients in $\mathbb{T}_r$. A Linear Combination $\mathtt{l}^r$ can indeed be thought of as a list of couples $(I, \tau_{\mathtt{l},I})_{I \in \mathbb{l}_\mathtt{l}}$ where $\mathbb{l}_\mathtt{l} \subset \mathbb{V}_h$ is the *stencil* described as a vector of global DOFs and $\tau_{\mathtt{l},I} \in \mathbb{T}_r$, $I \in \mathbb{l}_\mathtt{l}$, are the corresponding coefficients. The evaluation at $\mathbf{v}_h \in \mathbb{V}_h$ (obtained by calling the `operator()(`$\mathbf{v}_h$`)`) actually returns

$$\mathtt{l}^r(\mathbf{v}_h) = \sum_{I \in \mathbb{l}_\mathtt{l}} \tau_{\mathtt{l},I} v_I \in \mathbb{T}_r.$$

The concept havs been implemented with the following class:

```
template<typename ValueT,typename ItemT>
class LinearCombination {
```

```
public :

  //!evaluation for a MeshVariable
  template<typename VariableType>
  ValueT operator()(const VariableType& var) const ;

  //!List of integer ids of the combination mesh elements
  inline ConstArrayView<Integer> lids() const ;

  //!List of combination mesh elements
  inline ItemVectorViewT<ItemT> items() const {

  //!List of combination values
  inline ConstArrayView<ValueT> coefficients() const;
};
```

We have defined an efficient algebra extending in a straightforward manner the algebraic operators defined on the type `ValueT` of the linear combination coefficients. We have also defined for $\mathtt{l}^a_{I^1,(\tau^a_i)_{i\in I^1}}$ and $\mathtt{l}^b_{I^b,(\tau^b_i)_{i\in I^b}}$ the bilinear operations $\mathtt{l}^a * \mathtt{l}^b$ and $\mathtt{dot}(\mathtt{l}^a, \mathtt{l}^b)$ returning the local matrix $\mathbf{A}_{\text{loc}} = (m_{i,j})_{i\in I^b, j\in I^a}$ with $m_{i,j} = \tau_{a,j} * \tau_{b,i}$ respectively $\mathtt{dot}(\tau_{a,j}, \tau_{b,i})$.

In the algebra implementation, a particular care has been devoted to the computation of expressions containing the sum or subtraction of linear combinations, since this involves computing the intersection of the corresponding set of DOFs, see Figure 3.3. To overcome this difficulty, the linear combination algebra is implemented with the classes `LinearCombBuffer` and `LinearCombMng` with the template parameters `ItemT` and `ValueT`.

`LinearCombBuffer` implements the algebra operations `set()`, `add()`, **`mult()`** and `scaMul()`, and also the operators `+=`,`.*=` and `/=`.

`LinearCombMng` helps to manage sets of linear combinations with the same template parameters. It efficiently computes the union of linear combination stencils requested to implement the sum or substraction of linear combinations. The result of the built `LinearCombBuffer` can be stored in an instance of `LinearCombMng` that centralizes the memory management. Linear combinations can be used in computation as views on centralized data avoiding unuseful memory copy.

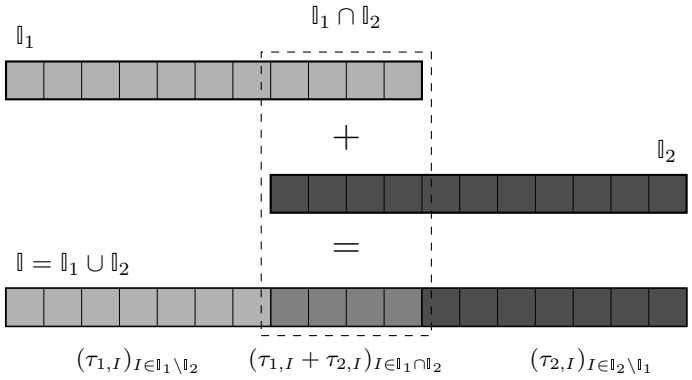


Figure 3.3: Computing the sum of two linear combinations $\mathtt{l}_1 = (I, \tau_{1,I})_{I\in\mathbb{I}_1}$ and $\mathtt{l}_2 = (I, \tau_{2,I})_{I\in\mathbb{I}_2}$ requires computing the intersection $\mathbb{I}_1 \cap \mathbb{I}_2$ and the union $\mathbb{I}_1 \cup \mathbb{I}_2$

This algebra allows to implement the gradient reconstruction operator $\mathfrak{G}_h^{\text{green}}$ defined by (2.10)

as described in Listing 3.3.

Listing 3.3: Implementation of the gradient reconstruction $\mathfrak{G}_h^{\mathrm{green}}$ (2.10) for an element $T \in \mathcal{T}_h$.

```
// define grad value type in a space of dimension dim
typedef Vector<ValueType,MeshType::dim> GradValueType ;
TraceOpType traceop(𝕍_h);// a given trace reconstruction operator
LinearCombT<ValueType,ItemType> vT(I_T,1.);
LinearCombBufferT<GradValueType,ItemType> buffer;
buffer.init()
for(F ∈ ℱ_T) {
  LinearCombT<ValueType,ItemType> vF = traceop(F);
  buffer.add(|F|_{d-1}/|T|_d n_{T,F},vF);
  buffer.add(-|F|_{d-1}/|T|_d n_{T,F},vT);
}
buffer.finalize();
LinearCombT<ValueType,ItemType> GT = buffer.linearComb();
```

The linear combination is a key ingredient in our framework upon which high level concepts such as function spaces, functions, test and trial functions implementation are built. Its implementation is therefore crucial to ensure good performance. Indeed the flexibility and efficiency of our framework is mainly based on the fact that the linear combination algebra enables partial lazy evaluation on linear combination expressions without evaluating the final result with the variable data values. This final evaluation can then be postponed and applied with several different variables at once.

Linear and bilinear contributions Exploiting the concept of linear combination, it is possible to devise a unified treatment for local contributions stemming from integrals over elements or faces. We illustrate the main ideas using the example: For a given $T \in \mathcal{T}_h$ and for u_h, $v_h \in V_h^{\mathrm{ccg}}$ we consider the local contribution $\mathbf{A}_{\mathrm{loc}}$ associated to the term

$$\int_T \kappa \nabla_h u_h \cdot \nabla_h v_h.$$

The key remark is that both $(\kappa\nabla_h u_h)|_T = \kappa|_T \nabla(u_h|_T)$ and $(\nabla_h v_h)|_T = \nabla(v_h|_T)$ are constant functions in T which can be represented as objects of type `LinearCombination<GradValueType,ItemType>`, say $\mathtt{l}_u = (\mathbb{J}, \tau_{\mathtt{l}_u,j})_{j\in\mathbb{J}}$ and $\mathtt{l}_v = (\mathbb{I}, \tau_{\mathtt{l}_v,i})_{i\in\mathbb{I}}$. The associated local contribution reads

$$\mathbf{A}_{\mathrm{loc}} = |T|_d \mathtt{dot}(\mathtt{l}_v, \mathtt{l}_u). \tag{3.1}$$

where $\mathtt{dot}(\mathtt{l}_v, \mathtt{l}_u)$ in our algebra returns a local matrix. Generalizing the above remark, one can implement local terms in matrix assembly as `BilinearContribution`s which can be represented by a local matrix $\mathbf{A}_{\mathrm{loc}}$ equal to the result of the bilinear operations $\mathtt{l}_v * \mathtt{l}_u$ or $\mathtt{dot}(\mathtt{l}_v, \mathtt{l}_u)$ multiplied by a factor γ. Observe, in particular, that $\mathbb{I}$ and $\mathbb{J}$ play the same role as the lines of the table of DOFs corresponding to test and trial functions supported in T in standard finite element implementations. As such, they are related to the lines and columns of the global matrix $\mathbf{A}$ to which $\mathbf{A}_{\mathrm{loc}}$ contributes,

$$\mathbf{A}(\mathbb{I}, \mathbb{J}) \leftarrow \mathbf{A}(\mathbb{I}, \mathbb{J}) + \mathbf{A}_{\mathrm{loc}}. \tag{3.2}$$

The additional argument γ in `BilinearContribution` serves as a multiplicative factor for the whole expression (in the above example, $\gamma = |T|_d$). More generally, γ can be a function of space and time, and may depend on discrete variables.

Similarly, right-hand side contributions can be represented by `LinearContributions`, which are initialized by a multiplicative coefficient γ and a linear combination $\mathtt{l}_v$. A typical assembly pattern is described in Listing 3.4.

Listing 3.4: Assembly of a bilinear and linear contribution (**A** represents here the global matrix **b** the global right-hand side vector)

```
LinearCombT<ValueType,ItemType> l_u, l_v;
// Assemble a bilinear contribution into the left-hand side
BilinearContribution<ValueType> blc(γ,LCAlgebra::dot(l_u, l_v));
A.assemble(blc);
// Assemble a linear contribution into the right-hand side
LinearContribution<ValueType> lc(γ,l_v);
b.assemble(lc);
```

3.2 Functional front-end

3.2.1 Function spaces

Incomplete broken polynomial spaces defined by (2.6) are mapped onto `C++` types according to the `FunctionSpace` concept detailed in Listing 3.5.

Listing 3.5: FunctionSpace concept

```
class FunctionSpace {
  // Types for trial and test functions
  typedef ... TrialFunctionType;
  typedef ... TestFunctionType;
  typedef ... FunctionType;
  // Create a new instance of the space
  FunctionSpace * create(const Mesh &);
  // Constant value of 𝔊_h|_S for S ∈ 𝒮_h expressed as a linear combination of DOFs
  LinearCombT<ValueT,ItemT> grad(S) const;
  // Value of ℜ_h|_S(x) for x ∈ S and S ∈ 𝒮_h expressed as a linear combination of DOFs
  LinearCombT<ValueT,ItemT> eval(S, x) const;
};
```

The actual types are generated by a helper template class **FunctionSpace** parametrized by a containing polynomial space, labeled **span**, and a piecewise constant gradient reconstruction, labeled **gradient** (labels for template arguments are here defined using the `boost::parameter` library).

The gradient reconstruction implicitly fixes both the vector space of DOFs $\mathbb{V}_h$ according to (2.6) as well as the choice (2.3) for $\mathcal{S}_h$. The helper template class **GradOp** enables to generate gradient reconstruction types for given values of the polynom degree, `SubmeshType`, `InterpolatorType` and `DOFType`, labeled respectively by the predefined keywords **poly**, **submesh**, **gradient**, **interpolator** and **dof**. `DOFType` are identified by the tags `space::tag::dofs::Th`, `space::tag::dofs::Fh` and `space::tag::dofs::ThxFh` representing respectively $\mathbb{R}^{\mathcal{T}_h}$, $\mathbb{R}^{\mathcal{F}_h}$ and $\mathbb{R}^{\mathcal{T}_h} \times \mathbb{R}^{\mathcal{F}_h}$.

The programming counterparts of the gradient reconstruction operator used in Chapter 2 are listed in Table 3.3.

The programming counterparts of the function spaces used in Chapter 2 are listed in Table 3.4.

For instance, in listing 3.6 we can see how is defined and generated a type representing a gradient reconstruction operator using the Green formula and a type representing a ccG Space.

Listing 3.6: ccG space definition

Table 3.3: **gradient** template parameters for the gradient reconstruction operator of Chapter 2

$\mathfrak{G}_h$	**submesh**	**interpolator**	**dof**
`GFormulaGradOp`	`tag::submesh::Ph`	`BarycentricOp`	`tag::dofs::Th`
`GreenFormulaGradOp`	`tag::submesh::Th`	`LInterpolatorOp`	`tag::dofs::Th`
`SUSHIFormulaHybridGradOp`	`tag::submesh::Ph`	–	`tag::dofs::ThxFh`

Table 3.4: **span** and **gradient** template parameters for the discrete spaces of Chapter 2

Space	$\mathcal{S}_h$	**span**	**gradient**
$\mathbb{P}_d^0(\mathcal{T}_h)$	$\mathcal{T}_h$	**poly**<0>	`Null`
V_h^{g}	$\mathcal{P}_h$	**poly**<1>	`GFormulaGradOp`
V_h^{ccg}	$\mathcal{T}_h$	**poly**<1>	`GreenFormulaGradOp`
V_h^{hyb}	$\mathcal{P}_h$	**poly**<1>	`SUSHIFormulaHybridGradOp`
V_h^{cc}	$\mathcal{P}_h$	**poly**<1>	`SUSHIFormulaGradOp`

```
typedef
GradOp< MeshType,                                          //!set mesh type
        submesh<tag::submesh::Th>,                         //!set submesh option
        interpolator<BarycentricOp>                        //!set trace operator
        dof<tag::dofs::Th>                                 //!set dof type
      >::type          GreenFormulaGradOp;

typedef
FunctionSpace< MeshType,                                   //!set mesh type
               span< poly<1>,                              //!set polynom degree
                     gradient<GreenFormulaGradOp >         //!set gradient type
                   >                                       //!generate space
             >::type CCGSpaceType;
```

We provide the user friendly free functions listed in Table 3.5 to create function space with default parameters.

Table 3.5: Function space fabric free functions

Space type	free function
$\mathbb{P}_d^0(\mathcal{T}_h)$	newP0Space($\mathcal{T}_h$)
V_h^{g}	newGSpace($\mathcal{T}_h$)
V_h^{ccg}	newCCGSpace($\mathcal{T}_h$)
V_h^{hyb}	newSUSHISpace($\mathcal{T}_h$)

The key role of a `FunctionSpace` is to bridge the gap between the algebraic representation of DOFs and the functional representation used in the methods of Chapter 2. This is achieved by the functions **grad** and **eval**, which are the `C++` counterpart of the linear operators $\mathfrak{G}_h$ and $\mathfrak{R}_h$ respectively; see §2.1. More specifically,

(i) for all $S \in \mathcal{S}_h$, **grad**`(`S`)` returns a vector-valued linear combination corresponding to the (constant) restriction $\mathfrak{G}_h|_S$;

(ii) for all $S \in \mathcal{S}_h$ and all $\mathbf{x} \in S$, **eval**`(`S`,` $\mathbf{x}$`)` returns a scalar-valued linear combination corresponding to $\mathfrak{R}_h|_S(\mathbf{x})$ defined according to (2.5).

The linear combinations returned by **grad** and **eval** can be used to build `LinearContributions` and `BilinearContributions` as described in the previous sections.

A function space type also defines the sub types `FunctionType`, `TestFunctionType` and `TrialFunctionType` corresponding to the mathematical notions of discrete functions, test and trial functions in variational formulations. Instances of `TrialFunctionType` and `FunctionType` are associated to a `Variable` object containing a vector of DOFs stored in memory associated to a string key corresponding to the variable name. For functions, the vector of dofs is used in the evaluation on a point $x \in \Omega$ while for trial functions, this vector is used to receive the solution of the discrete problem. Test functions representing implicitely the space basis, are not associated to any `Variable` objects, neither vector of dofs. Unlike `FunctionType`, the evaluation of `TrialFunctionType` and `TestFunctionType` is lazy in the sense that it returns a linear combination. This linear combination can be used to build local linear or bilinear contributions to the global system, or enables to postpone the evaluation with the variable data.

3.2.2 Gradient reconstruction operator

An Interpolator operator $\mathfrak{T}_h : \mathbb{V}_h \to \mathbb{R}^{\mathcal{F}_h}$ realizes the mapping $\mathbb{V}_h \ni \mathbf{v}_h \mapsto \mathfrak{T}_h(\mathbf{v}_h) = (v_F)_{F\in\mathcal{F}_h}$ with $(v_F)_{F\in\mathcal{F}_h} \in \mathbb{R}^{\mathcal{F}_h}$ and, for all $F \in \mathcal{F}_h$,

$$v_F = \langle \xi^{\mathfrak{g}_F}_{\mathbf{v}_h} \rangle_F = \xi^{\mathfrak{g}_F}_{\mathbf{v}_h}(\overline{\mathbf{x}}_F). \tag{3.3}$$

Most of Interpolator operators can be set defining for each $F \in \mathcal{F}_h$ the linear combination L_F such that $v_F = L_F(\mathbf{v}_h)$.

The interpolator concept specifies a classe type that implements the function:
`LinearCombT<Cell,ValueType> eval(Face const & face)` that returns L_F for each $F \in \mathcal{F}_h$.

```
class InterpolatorType {
 public:
 typedef MeshType::Face Face;
 typedef MeshType::Cell Cell;
 LinearCombT<Cell,ValueType> eval(Face const & face) const;
};
```

The mapping $v_F = L_F(\mathbf{v}_h)$ is then realized evaluating `vf` on each `face` of a `mesh` as follows:

```
LinearCombT<Cell,ValueType> comb = interpolator.eval(face);
ValueType vf = comb(face);
```

We have implemented the L-Interpolator that uses the L-construction procedure with a piecewise constant tensor field on $\mathcal{T}_h$, and the Barycentric Interpolator that builds for each $F \in \mathcal{F}_h$ a group $S_F \subset \mathcal{T}_h$ of cell neighbours and computes the barycentric coordinates $(\alpha_T)_{T\in S_F}$ of the barycenter $x_F = \sum_{T\in S_F} \alpha_T x_T$ in the barycenters system $(x_T)_{T\in S_F}$.

We have seen that for a given mesh $\mathcal{T}_h$, a gradient reconstruction operator $\mathfrak{G}_h$ fixes the submesh $\mathcal{S}_h$ and the value of $\mathfrak{G}_h(\mathbf{v}_h)|_T \in \mathbb{R}^d$ for each $\mathbf{v}_h \in \mathbb{V}_h, T \in \mathcal{S}_h$. It can be set defining for each $T \in \mathcal{S}_h$ a linear combination L_T, such that $\mathfrak{G}_h(\mathbf{v}_h)|_T = L_T(\mathbf{v}_h)$ for each $\mathbf{v}_h \in \mathbb{V}_h$. The **Gradient operator concept** specifies classes that define `SubMeshType` precising the submesh type on which gradient is piecewise constant and that implement an evaluation function
`LinearCombT<Cell,VectorValueT> eval(`$S \in \mathcal{S}_h$`)` on each submesh element. The signature of the function depends of $\mathcal{S}_h$ conforming to the way submesh elements are identified in §3.1.1.

`template<typename InterpolatorT> class GreenFormulaGradOpT` implements the gradient operator $\mathfrak{G}_h(.)$ based on an interpolator operator $\mathfrak{T}_h(.)$ that builds the gradient on cell elements using the green formula:

$$\mathfrak{G}_h(\mathbf{v}_h)|_T = \frac{1}{|T|_d} \sum_{F\in\mathcal{F}_T} |F|_{d-1}(v_F - v_T)\mathrm{n}_{T,F}$$

For $T \in \mathcal{T}_h$, $F \in \mathcal{F}_T$, let $L_F^T = \mathfrak{T}_h(.)|_F$ be the linear combination value of $\mathfrak{T}_h(.)$ on F, L_T^I the identity linear combination on T, and L_T^G the value of gradient operator on T. Using $v_F = L_F^T(\mathbf{v}_h)$ we have $\mathfrak{G}_h(\mathbf{v}_h)|_T = L_T^G(\mathbf{v}_h) = \frac{1}{|T|_d} \sum_{F \in \mathcal{F}_T} |F|_{d-1}(L_F^T(\mathbf{v}_h) - L_T^I(\mathbf{v}_h))\mathrm{n}_{T,F}$.

The gradient operator is built setting for each T $L_T^G = \frac{1}{|T|_d} \sum_{F \in \mathcal{F}_T} |F|_{d-1}(L_F^T - L_T^I)\mathrm{n}_{T,F}$

3.2.3 Linear and bilinear forms

The **BilinearForm** and **LinearForm** concepts represent the linear and bilinear forms described in Chapter 2. They allow to define expressions using test and trial functions, unary and binary operators. Then, using the linear and bilinear contribution concepts defined in 3.1.3 and the matrix and vector types of the linear system framework, they allow to write integration algorithms (listing 3.7) leading to the construction a global linear system, based on an assembly procedure that adds local terms computed on each elements of the mesh.

Listing 3.7: Integration algorithm to evaluate a linear and bilinear form

```
MeshType Th;          // declare Th
SpaceType Uh(Th);     // Trial function space
SpaceType Vh(Th);     // Test function space
SpaceType::TrialFunctionType u(Uh);
SpaceType::TestFunctionType v(Vh);
LinearAlgebra::Matrix matrix(Uh,Vh);
LinearAlgebra::Vector rhs(Vh);
std::for_each(All<Cell>::items(Th).begin(),
              All<Cell>::items(Th).end(),
              [& val](Cell& cell)
{
  ValueType meas = measure(Th, cell);
  BilinearContribution<ValueType> GuGv =
   LinearCombAlgebra::dot(grad(u).eval(cell),grad(v).eval(cell));
  matrix.assemble(meas,GuGv);
  BilinearContribution<ValueType> uv =
   LinearCombAlgebra::mult(id(u).eval(cell),id(v).eval(cell))
  matrix.assemble(meas,uv);
  rhs.assemble(meas*f[cell],id(v).eval(cell));
}
```

Generalizing Example 3.1.3, we notice that bilinear forms result from the sum of terms with the following general form:

$$\sum_{I \in \mathcal{I}_h} \int_I (\gamma_u \times \mathcal{L}_u(u_h)) \cdot (\gamma_v * \mathcal{L}_v(v_h)), \tag{3.4}$$

where

(i) $\mathcal{I}_h \in \{\mathcal{T}_h, \mathcal{F}_h, \mathcal{F}_h^i, \mathcal{F}_h^b\}$ is a set of mesh items (cf. §3.1);

(ii) γ_u and γ_v are tensor fields of rank r_{γ_u} and r_{γ_v} respectively possibly depending on constants and on discrete variables;

(iii) $\mathcal{L}_u$ is a linear operator acting on the trial function $u_h \in U_h$ and yielding a tensors-valued field of order r_u. The operator $\mathcal{L}_u$ is represented by an instance of `LinearCombination`;

(iv) $\mathcal{L}_v$ is a linear operator acting on the test function $v_h \in V_h$ (which can possibly belong to a space $V_h \neq U_h$) and yielding a tensor-valued field of order r_v. The operator $\mathcal{L}_v$ is represented by an instance of `LinearCombination`;

(v) $\times$ (resp. $*$) is an admissible product between a tensor of order r_{γ_u} (resp. r_{γ_v}) and a tensor of order r_u (resp. r_v) yielding a tensor-valued field with order r;

(vi) $\cdot$ is the contraction product for tensors of order r.

The factors $(\gamma_u \times \mathcal{L}_u(u_h))$ and $(\gamma_v * \mathcal{L}_v(v_h))$ are respectively referred to as a *trial* and *test expression*.

Example 1 (Bilinear term). The term considered in Example 3.1.3 can be recast into the form (3.4) by setting $\mathcal{I}_h = \mathcal{T}_h$, $\gamma_u = \kappa$ and $r_{\gamma_u} = 2$, $\gamma_v = 1$ and $r_{\gamma_v} = 0$, $\mathcal{L}_u = \nabla_h$ and $r_u = 1$, $\mathcal{L}_v = \nabla_h$ and $r_v = 1$, and denoting by $\times$, $*$, and $\cdot$ the standard matrix-vector product, the scalar product, and the standard vector inner product respectively.

Right-hand side contributions can be handled in a similar fashion.

3.3 DSEL design and implementation

The main goal of the DSEL is to allow a notation as close as possible to that of Chapter 2. The focus of this section is on bilinear forms, as the ingredients for linear forms are essentially similar. In what follows we do not mean to be exhaustive. Instead we first define our DSEL giving the production rules that enable to create trial and test expressions as well as bilinear terms of the form (3.4) using the Extended Backus–Naur Form (EBNF), [19], then we detail how this DSEL has been implemented using the tools provided by the Boost Proto framework.

3.3.1 Language definition

Terminals and keywords The terminals of the DSEL is composed of a number of predefined types categorized in the following families:

- the `BaseType` family for the standard `C++` types representing integers and reals;
- the `VarType` family for all discrete variable types defined in §3.1;
- the `MeshGroupType` family for types representing collections of mesh entities such as the ones listed in Table 3.1;
- the `DiscreteFunction`, `TestFunction` and `TrialFunction` families representing the discrete functions, test and trial functions defined in §3.2.

The DSEL is based on some predefined keywords listed in table 3.6 semantically close to their counterpart in the mathematical framework.

Trial and test expressions Trial (resp. test) expressions are obtained as the product of a coefficient γ_u (resp. γ_v) by a linear operator $\mathcal{L}_u$ (resp. $\mathcal{L}_v$) acting on a trial (resp. test) function. The coefficient can result from the algebraic combination of constant values and `Variable`s evaluated at item I (cf.(3.4)). Listing 3.8 defines the production rules that enable to create coefficient expressions involving, in particular, constant values, `Variable`s over `Cell`s and products thereof.

Listing 3.8: Examples of production rules for the coefficient γ in (3.4)

```
BaseExpr = BaseType | BaseExpr * BaseExpr;

VarExpr  = VarType | BaseExpr * VarExpr | VarExpr * VarExpr;

CoefExpr = BaseExpr | VarExpr;
```

To obtain trial and test expressions, we introduce linear operators acting on test and trial functions. A few examples are provided in Listing 3.9, and include (i) **grad**, the gradient of the trial/test function; (ii) trace operators like **jump** and **avg** representing, respectively, the jump

and average of a trial/test function across a face. Besides linear operators, the production rules for trial and test expressions in Listing 3.9 include various products by coefficients resulting from the production rules of Listing 3.8 (**dot** denote the vector inner product).

Listing 3.9: Production rules for trial and test expressions

```
LinearOperator = "grad" | "jump" | "avg";

TrialExpr = TrialFunction                  |
            CoefExpr * TrialExpr           |
            "dot("CoefExpr, TrialExpr")"   |
            LinearOperator"("TrialExpr")";

TrialExpr = TestFunction                   |
            CoefExpr * TestExpr            |
            "dot("CoefExpr, TestExpr")"    |
            LinearOperator"("TestExpr")";
```

Bilinear forms Once test and trial expressions are available, bilinear terms can be obtained as contraction products of trial and test expressions or as sums thereof, as described in Listing 3.10.

Listing 3.10: Production rules for bilinear terms

```
BilinearTerm = TrialExpr * TestExpr          |
               "dot("TrialExpr, TestExpr")"  |
               CoefExpr * BilinearTerm       |
               BilinearTerm + BilinearTerm;
```

Bilinear forms finally result from the integration of bilinear terms on groups of mesh items (cf. Table 3.2). Production rules for bilinear forms are given in Listing 3.11. Observe that **integrate** acts as a binary operator that takes as arguments the group of items over which integration is performed and the bilinear term to integrate.

Listing 3.11: Production rules for bilinear forms

```
IntegrateBilinearTerm = "integrate("MeshGroup, BilinearTerm")";
BilinearForm = IntegrateBilinearTerm                 |
               IntegrateBilinearTerm + BilinearForm;
```

3.3.2 Language design and implementation with `Boost.Proto`

We have based our implementation on the `Boost.Proto` library by Niebler [83], a powerful framework to build DSELs in C++. In the online documentation[83], the framework is presented as follows:

> This library provides a collection of generic concepts and metafunctions that help to design a DSL, its grammar and tools to parse and evaluate expressions. It provides tools for constructing, type-checking, transforming and executing expression templates [28, 34, 92], more specifically, it provides: (i) an expression tree data structure, (ii) a mechanism for giving expressions additional behaviors and members, (iii) operator overloads for building the tree from an expression, (iv) utilities for defining the grammar to which an expression must conform, (v) an extensible mechanism for immediately executing an expression template, (vi) an extensible set of tree transformations to apply to expression trees.

This framework enables to design a DSEL in a declarative way with mechanisms based on concepts like: (i) *tag*, (ii) *meta-function*, (iii) *grammar*, (iv) *context*, (v) and *transform* structures (see the framework documentation [83] for more details).

In this section, we detail in the first part how we have translated our language formal definition §3.3.1 in proto objects that enable to define expressions, the language grammar, *context* and *transforms* structures to evaluate expressions and implement algorithms. We present first the mechanisms used to create expressions with user types, and the tools to parse and introspect them. We introduce then the *specific domain* structure that allow us to encapsulate all the expressions that one user can define. We show how, using *grammar* structures we can define constraints on expressions and match patterns in them. We explain finally how we can extend the language with new keywords and associate them to *grammar* structures. In the last part, we show how we have written algorithms by evaluating expressions with *context* or *transform* objects.

Language front-ends

The language front ends are defined by: (i) the terminals; (ii) the keywords listed in 3.6; (iii) and the grammar based on the production rules of Listings 3.8, 3.9, 3.10, and 3.11. Expressions are implemented with proto expression tree structures where each node is an object of type `proto::base_expr` identified by a tag and where the leafs of the tree are occupied by terminals (cf. Listing 3.8), meshes (cf. Listing 3.11), test and trial functions (cf. Listing 3.9).

Example 2 (Bilinear form for the SUSHI method). The programming counterpart of the bilinear form a_h^{sushi} defined by (2.15) is given in Listing 3.12. The corresponding expression tree is detailed in Fig. 3.4.

Listing 3.12: DSEL based implementation of the bilinear form a_h^{sushi} defined by (2.15)

```
1 Mesh Th(/* ... */);
2 auto Vh = newSUSHISpace(Th);
3 auto uh = Vh->trial();
4 auto vh = Vh->test();
5 // Observe that the language automatically handles the fact that gradients are piecewise constant
6 // over pyramids rather than cells
7 BilinearForm ah = integrate(allCells(Th),
8                              dot(K*grad(uh),grad(vh));
```

Tag structures and meta functions

The implementation of a proto expression tree is based on tag structures and on associated meta-functions that enable to create nodes, implement grammar or transform structures.

The `boost::proto` framework already provides standard tags for standard unary and binary C++ operator and metafunctions to easily navigate in the expression tree (cf table 3.7).

We have completed them with tags representing : (i) the different types of the DSEL terminals (the leafs of the tree) ; (ii) the DSEL keywords corresponding to the nodes of the tree.

Listing 3.13: Tags definition

```
namespace fvdsel {
  namespace tag {
    //! DSEL terminal tags
    struct basetype{} ;
    struct meshvartype{} ;
    struct testfunctiontype{} ;
    struct trialfunctiontype{} ;
    struct meshzonetype{} ;
```

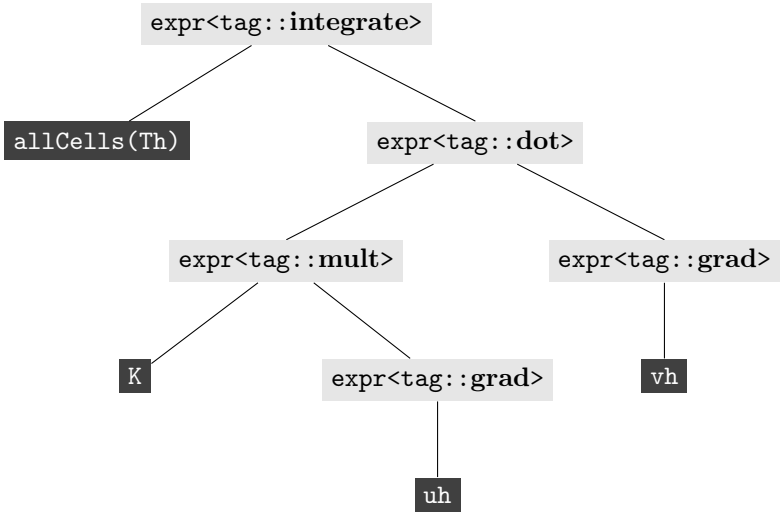


Figure 3.4: Expression tree for the bilinear form defined at line 8 of Listing 3.12. Expressions are in *light gray*, language terminals in *dark gray*

```
    struct nulltype{} ;

    //! DSEL keyword tags
    struct dot{} ;
    struct grad{} ;
    struct jump{} ;
    struct avg{} ;
    struct integrate{} ;
    }
}
```

FVDSL domain definition We have defined our domain `FVDSLDomain` where all expressions are encapsulated in a `FVDSLExpr` that comform to our grammar `FVDSLGrammar` detailed in §3.3.2. This mechanism enables the `Boost.Proto` framework to overload most of C++ operators.

Listing 3.14 illustrates how we have defined our domain.

Listing 3.14: FVDSL expression domain definition

```
template<typename Expr> struct FVDSLExpr;

struct FVDSLGrammar
  : proto::or_<
        proto::terminal<boost::proto::_>
      , proto::nary_expr<boost::proto::_, proto::vararg<FVDSLGrammar> >
    >
{};

// Expressions in the pde domain will be wrapped in FVDSLExpr<>
// and must conform to the FVDSLGrammar
struct FVDSLDomain
  : proto::domain<proto::generator<FVDSLExpr>, FVDSLGrammar>
{};
```

```
template<typename Expr>
struct FVDSLExpr
  : proto::extends<Expr, FVDSLExpr<Expr>, FVDSLDomain>
{
    explicit FVDSLExpr(Expr const &expr)
      : proto::extends<Expr, FVDSLExpr<Expr>, FVDSLDomain>(expr)
    {}

    BOOST_PROTO_EXTENDS_USING_ASSIGN(FVDSLExpr)

};
```

DSEL keywords The DSEL keywords listed in table 3.6 are associated to specific tags. For each tag, we have implemented a free function that creates a tree node associated to this tag, a meta-function that generates the type of that node, and a grammar element that matches expressions and as *PrimitiveTransform* (cf [83], 3.3.2) that dispatches to the `proto::pass_through<>` transform. For instance, the **grad** keyword is associated to the tag `fvdsel::tag::grad`. Listing 3.15 illustrates the definition of the unary free function **grad(.)** creating nodes with that tag and the definition of `fvdsel::gradop<ExprT>` the meta-function that matches grad expression or dispatches transforms. The **dot** keyword is associated to the tag `fvdsel::tag::dot`. Listing 3.16 illustrates the definition of the binary free function **dot(.,.)** creating nodes with that tag and the definition of `fvdsel::dotop<LExprT,RExprT>` the meta-function that matches inner product expression or dispatches transforms.

Listing 3.15: free function and meta-function associated to fvdsel::tag::grad

```
//! grad metafunction
template<typename A>
typename proto::result_of::make_expr< fvdsel::tag::grad , FVDSLDomain
                                      , A const &
                                    >::type
grad(A const &a)
{
    return proto::make_expr<fvdsel::tag::grad, FVDSLDomain>(boost::ref(a));
}

//! grad metafunction
template<typename T>
struct gradop :  proto::transform< gradop<T> >
{
  // types
  typedef proto::expr< fvdsel::tag::grad,
                       proto::list1< T >
                     >                                    type;
  typedef proto::basic_expr< fvdsel::tag::grad,
                             proto::list1< T >
                           >                              proto_grammar;

  // member classes/structs/unions
  template<typename Expr, typename State, typename Data>
  struct impl :
     proto::pass_through<gradop>::template impl<Expr, State, Data>
  {
  };
};
```

Listing 3.16: Free function and meta-function associated to fvdsel::tag::dot

```
template<typename L,typename R>
typename
proto::result_of::make_expr<
                              fvdsel::tag::dot
                              , FVDSLDomain
                              , L const &
                              , R const &
                             >::type
dot(L const &l,R const& r)
{
  return proto::make_expr< fvdsel::tag::dot,
                           FVDSLDomain >(boost::ref(l),boost::ref(r));
}

template<typename LeftT,typename RightT>
struct dotop :  proto::transform< dotop<LeftT,RightT> >
{
  // types
  typedef proto::expr< fvdsel::tag::dot,
                       proto::list2< LeftT,RightT >
                     >                                              type;
  typedef proto::basic_expr< fvdsel::tag::dot,
                             proto::list2< LeftT, RightT >
                            >                                       proto_grammar;

  // member classes/structs/unions
  template<typename LExpr, typename RExpr, typename State, typename Data>
  struct impl :
     proto::pass_through<dotop>::template impl<LExpr,RExpr, State, Data>
  {
  };
} ;
```

Table 3.5 lists the main keywords with their associated tags, free functions and meta-functions.

Figure 3.5: DSEL keywords

keyword	n-arity	tag	free function	meta-function
integrate	2	`fvdsel::tag::integrate`	**integrate**(.,.)	`integrateop<.,.>`
grad	1	`fvdsel::tag::grad`	**grad**(.)	`gradop<.>`
jump	1	`fvdsel::tag::jump`	**jump**(.)	`jumpop<.>`
avg	1	`fvdsel::tag::avg`	**avg**(.)	`avgop<.>`
dot	2	`fvdsel::tag::dot`	**dot**(.,.)	`dotop<.,.>`

Grammar definition :

The grammar of our language is based on the production rules detailed in §3.3.1. Proto provides a set of tools that enables us to implement each production rule in a user friendly declarative way. Terminal structures are detected with the meta-function defined in listing 3.17. Each production rule is implemented by a grammar structure composed with other grammar structures, proto pre-defined transforms (cf table 3.7) or some of our specific transforms (cf table 3.5).

Listing 3.17: terminal meta-function

```
template<typename T> struct is_base_type ;
```

```
template<typename T> struct is_mesh_var ;
template<typename T> struct is_mesh_group ;
template<typename T> struct is_function ;
template<typename T> struct is_test_function ;
template<typename T> struct is_trial_function ;

template<typename T>
struct IsFVDSLTerminal
  : mpl::or_<
              fvdsel::is_function_type<T>,
              fvdsel::is_base_type<T>,
              fvdsel::is_mesh_var<T>,
              fvdsel::is_mesh_group<T>
            >
{};
```

In listing 3.18 we can compare the implementation of the DSEL grammar with the `BaseTypeGrammar`, `MeshVarTypeGrammar`, `TestFunctionTerminal`, `TrialFunctionTerminal`, `CoefExprGrammar` and `BilinearGrammar` structures to the EBNF definition of the production rules 3.8, 3.9, 3.10, and 3.11 specifying bilinear expressions.

Listing 3.18: Bilinear expression grammar

```
namespace fvdsel {

  struct BaseTypeGrammar
  : proto::terminal< proto::convertible_to<Real> >
  {} ;

  struct MeshVarTypeGrammar
  : proto::and_< proto::terminal<proto::_>,
                 proto::if_< fvdsel::is_mesh_var<proto::_value>() > >
  {} ;

  struct TestFunctionTerminal
  : proto::and_< FunctionTerminal,
                 proto::if_< fvdsel::is_test_function<proto::_value>() > >
  {} ;

  struct TrialFunctionTerminal
  : proto::and_< FunctionTerminal,
                 proto::if_< fvdsel::is_trial_function<proto::_value>() > >
  {} ;

  struct CoefExprGrammar ;

  struct CoefExprGrammar
  : proto::or_<
              BaseTypeGrammar,
              MeshVarTypeGrammar,
              proto::plus<CoefExprGrammar, CoefExprGrammar>,
              proto::multiplies<CoefExprGrammar, CoefExprGrammar>,
              proto::divides<CoefExprGrammar, CoefExprGrammar>
            >
  {} ;

  struct TrialExprGrammar
    : proto::or_< TrialFunctionTerminal,
                  proto::multiplies<CoefExprGrammar, TrialExprGrammar>,
```

```
                    fvdsel::jumpop<TrialExprGrammar>,
                    fvdsel::avgop<TrialExprGrammar>,
                    fvdsel::gradop<TrialExprGrammar>,
                    fvdsel::traceop<TrialExprGrammar>
                  >

  {} ;

  struct TestExprGrammar
    : proto::or_< TestFunctionTerminal,
                  proto::multiplies<CoefExprGrammar, TestExprGrammar>,
                  fvdsel::jumpop<TestExprGrammar>,
                  fvdsel::avgop<TestExprGrammar>,
                  fvdsel::gradop<TestExprGrammar>,
                  fvdsel::traceop<TestExprGrammar>
                >
  {} ;

  struct BilinearGrammar ;

  struct PlusBilinear
    : proto::plus< BilinearGrammar, BilinearGrammar >
   {};

   struct MinusBilinear
    : proto::minus< BilinearGrammar, BilinearGrammar >
   {};

  struct MultBilinear
    : proto::multiplies< CoefExprGrammar, BilinearGrammar >
  {};

 struct BilinearGrammar
 : proto::or_<
              proto::multiplies<TrialExprGrammar, TestExprGrammar>,
              fvdsel::dotop<TrialExprGrammar, TestExprGrammar>,
              PlusBilinear,
              MinusBilinear,
              MultBilinear
            >
 {} ;
}
```

Context evaluation and transforms

Language back-ends: Expression evaluation, algorithm implementation The DSEL back-ends are composed of algebraic structures (matrices, vectors, linear combinations) used in different kinds of algorithms based on iterations on mesh entities, matrices, vectors evaluation or assembly operations. Theses algorithms are implemented by evaluating and manipulating our `FVDSLDomain` expressions. Such evaluations are based on two kind of Proto concepts: *Contexts* and *Transforms* structures. In [83] these concepts as presented as follows:

- A *Context* is like a function object that is passed along with an expression to the proto::eval() function. It associates behaviors with node types. `proto::eval()` walks the expression and invokes your context at each node.
- A *Transform* is a way to associate behaviors, not with node types as in an expression, but with rules in a Proto grammar. They are like semantic actions in

other compiler-construction toolkits.

Algorithms are then implemented as specific expression tree evaluations, as a sequence of piece of algorithms associated to the behaviour of `Evaluation Context` on each node or on transforms that match production rules.

For instance, let us consider the bilinear form defined by the following expression:

Listing 3.19: SUSHI bilinear form label

```
BilinearForm ah = integrate(allCells(Th), dot(K*grad(u),grad(v)) ;
```

`allCells(Th)`, `K`, `u`, `v` are terminals of the language. **integrate**, **dot** and **grad** are specific keywords of the language associated to the tags `fvdsel::tag::`**integrate**, `fvdsel::tag::`**dot** and `fvdsel::tag::`**grad**. The binary operator `*` is associated to the tag `proto::tag::`**mult**

At evaluation, the expression is analyzed as follows:

1. The root node of the tree is associated to the tag `tag::`**integrate** composed of an `MeshGroup` expression (`allCells(Th)`) and the `BilinearTerm` expression (**dot**`(K*`**grad**`(u),`**grad**`(v))`);
2. The integration algorithm consists in iterating on the cell elements of the `allCells(Th)` and evaluating the bilinear expression on each cell. This bilinear expression is composed of:
   - a `TrialExpr` expression: `K*`**grad**`(u)`;
   - a `TestExpr` expression: **grad**`(v)`
   - a binary operator associated to the tag: `tag::`**dot**

   The evaluations of the `TrialExpr` expression and of the `TestExpr` expression on a cell return two linear combination objects which, associated to the binary operator tag lead to a bilinear contribution which is a local matrix contributing to the global linear system of the linear context with a factor equal to the measure of the cell.

To implement the integration algorithm associated to linear variational formulation, we have used both *Context* and *Transform* structures. A `BilinearContext` object, referencing a linear system back-end object used to build the global linear system with different linear algebra packages has been developed to evaluate the global expression. On an Integrate node, this object calls a `IntegratorOp` transform on the expression tree. In listing 3.3.2, we detail the implementation of this transform that matches in our example the expression with the tag `fvdsel::tag::`**integrate**, the `MeshGroup` expression `allCells(Th)` and the term **dot**`(K*`**grad**`(u),`**grad**`(v))`.

```
struct Integrator : proto::callable
{
  //... callable object that will use a BilinearIntegrator transform on
  // a bilinear expression
  typedef int result_type;

  template<typename ZoneT,typename ExprT, typename StateT, typename DataT>
  int
  operator()(ExprT const& expr,ZoneT const& zone,StateT& state, DataT const& data) const
  {
      //call a transform that analyze ExprT and dispatch to the appropriate
      //transform
      return 0 ;
  }
} ;

struct IntegratorOp
```

```
: proto::or_<
            proto::when<
                        fvdsel::IntegratorGrammar,
                        fvdsel::Integrator(proto::_child_c<2>, //! expr
                                           proto::_child_c<1>, //! zone
                                           proto::_state,      //! state
                                           proto::_data        //! context
                                           )
                       >
            proto::when<
                        proto::plus<IntegratorOp,IntegratorOp>,
                        IntegratorOp(proto::_left,
                                     IntegratorOp(proto::_right,
                                                  proto::_state,
                                                  proto::_data
                                                 ),
                                     proto::_data)
                       >
             >
  {};
```

In the `Integrator` callable transform, analyzing the integrate expression term, when a bilinear expression is matched, another transform `BilinearIntegrator` (listing 3.3.2) matching a `DotExpr` associated to `fvdsel::tag::dot` and the production rules matching the test and trial part of the bilinear expressions. The algorithm (listing 3.20) is called by the callable transform `DotIntegrator`. Note that the `BilinearContext` is passed along the expression tree with the `proto::_data` structure.

```
struct MultIntegrator : proto::callable
{
  typedef int result_type;
  template<typename TrialExprT,
           typename TestExprT,
           typename StateT,
           typename DataT>
  int
  operator()(TrialExprT const& lexpr,
             TestExprT const& rexpr,
             StateT& state,
             DataT const& data) const
  {
    // call integrate algorithm
    // with tag proto::tag::mult

    return integrate<proto::tag::mult>(getMesh(data),
                                       getGroup(data),
                                       lexpr,
                                       rexpr,
                                       GetContext(data) ) ;
  }
} ;

struct DotIntegrator : proto::callable
{
  typedef int result_type;
  template<typename TrialExprT,
           typename TestExprT,
           typename StateT,
```

```
            typename DataT>
  int
  operator()(TrialExprT const& lexpr,
             TestExprT const& rexpr,
             StateT& state,
             DataT const& data) const
  {
    // call integrate algorithm
    // with tag proto::tag::dot
    return integrate<proto::tag::dot>(getMesh(data),
                                      getGroup(data),
                                      lexpr,
                                      rexpr,
                                      GetContext(data) ) ;
    }
} ;

struct BilinearIntegrator
: proto::or_<
              proto::when< proto::multiplies<TrialExprGrammar,TestExprGrammar>,
                           MultIntegrator(proto::_left,          //! lexpr
                                          proto::_right,         //! rexpr
                                          proto::_state,         //! state
                                          proto::_data           //! context
                                          )>,
              proto::when< fvdsel::dotop<TrialExprGrammar,TestExprGrammar>,
                           DotIntegrator(proto::_child_c<0>, //! left
                                         proto::_child_c<1>, //! trial
                                         proto::_state,      //! state
                                         proto::_data        //! context
                                        )
                         >,
              proto::when< proto::plus<BilinearGrammar,BilinearGrammar>,
                           BilinearIntegrator(proto::_right,//! bilinear expr
                                              BilinearIntegrator(proto::_left,
                                                                 proto::_state,
                                                                 proto::_data),
                                              proto::_data    //! context
                                             )
                         >
            >
{} ;
```

Listing 3.20 is a simple assembly algorithm. We iterate on each entity of the mesh group and evaluate the test and trial expressions on each entity. For this evaluation, we have defined different kinds of context objects. The structure `EvalContext<ItemT>` enables to compute the linear combination objects that return test and trial expressions, which associated to the binary operator tag lead to a bilinear contribution, a local matrix contributing to the global linear system of the linear context with a factor equal to the measure of the cell. Note that the `BilinearContextT` is parametrized by a `phase_type` parameter that enables to optimize and factorize the global linear system construction: intermediate computation can be stored in system cache and be reused. For instance when a global linear system is built, the global system dimensions setting phase, the sparse structure matrix definition and the matrix filling phase can be separated. The first two phases can be easily factorized for several filling phases in iterative algorithms.

Listing 3.20: Integration assembly algorithm

```
template<typename ItemT,
         typename TestExprT,
         typename TrialExprT,
         typename tag_op,
         typename BilinearContextT>
void integrate(Mesh const& mesh,
               GroupT<ItemT> const& group,
               TrialExprT const& trial,
               TestExprT const& test,
               BilinearContextT& ctx)
{
  static const Context::ePhaseType phase = BilinearContextT::phase_type;
  auto matrix = ctx.getMatrix();
  for( auto cell : group )
  {
    EvalContext<Item> ctx(cell) ;                    //! eval context on mesh item
    auto lu = proto::eval(trial,ctx) ;               //! trial linear combination
    auto lv = proto::eval(test,ctx));                //! test linear combination
    BilinearContribution<tag_op> uv(lu,lv) ;
    assemble<phase>(matrix,                          //! matrix
                    measure(mesh,cell),              //! cell measure
                    uv ) ;                           //! bilinear contribution


  }
}
```

In the same way the evaluation of a linear form expression with a linear context leads to the construction of the right hand side of a global linear system.

Once built, the global linear system can be solved with a linear system solver provided by the linear algebra layer.

3.3.3 Extensions for vectorial expressions

Vector functions are used in many problems. They represent 2D and 3D fields for instance (velocity fields) or properties of collection of entities. To have a more expressive language, the grammar has been extended to express loops on vectorial expression components with the following tools:

(i) the concept of `rank` has been introduced to qualify the expression rank (`Scalar`, `Vector` or `Tensor`);

(ii) The classes `FunctionArray`, `TestFunctionArray` and `TrialFunctionArray` representing vectors elements $(u_1, \ldots, u_n) \in \mathbb{V}_h \times \ldots \times \mathbb{V}_h = \mathbb{V}_h^n$, are new terminals used to build vectorial expressions;

(iii) The new concepts `Range` and `Index` enable to iterate on Vector or Tensor expressions. The `Range` is associated to a finite number of indices (i,j,k,...) and provides iterators for each of them. We have 1D `Range` with one index for vectorial expression and 2D `Range` with two indexes, for tensorial expression.

We have completed our language definition with:

- the new terminals `Range` and `Index`;
- the new keywords `sum` defining an unary function `sum(<range>)` that enables to iterate on the indexes of a range, `Dxi` defining the binary function `Dxi(<index>,<expr>)` giving the component values of an **grad**`(<expr>)` expression;

- the new production rules adding the `operator()(<index>)` to vectorial and tensorial terminals and the `operator[](<expr>)` to `sum(<range>` nodes.

The Proto framework enables to overload the `operator()` and `operator[]` creating expression nodes associated to the tags `proto::tag::function` and `proto::tag::subscript`. It is then possible to evaluate separately each componant of vectorial and tensorial expressions iterating on the index values. For instance, with vector variables with the dimension of the mesh, the two following expressions:
`sum(_i)[ dxi(_i,u(_i)) ]` and **div**`(u)` are equivalent.

The mathematical expressions: $a_1(\mathbf{u},\mathbf{v}) \stackrel{\text{def}}{=} \int_\Omega \nabla\mathbf{u}{:}\nabla\mathbf{v}$ and $a_2(\mathbf{u},\mathbf{v}) \stackrel{\text{def}}{=} \int_\Omega \Sigma_{i,j}\partial_j u_i v_j$ have the following programming countpart:

```
static const int dim = 3;
// declare i,j in 0,1,2
Range<2> range(dim,dim);
Range<2>::Index& _i = range.get<0>();
Range<2>::Index& _j = range.get<1>();
// declare u and v
auto u = Uh->trialArray("U",dim);
auto v = Uh->testArray("V",dim);

BilinearForm a1 =
  integrate(allCells(Th),
  sum(_i) [ dot(grad(u(_i)),grad(v(_i))) ];

BiLinearForm a2 =
  integrate(allCells(mesh),
  sum(range)[ dxi(_j,u(_i)) * v(_j) ) ];
```

3.3.4 Extensions for boundary conditions management

In Chapter 2 we have presented only homogeneous boundary conditions. In fact most of these methods are easily extended to more general boundary conditions. Let $\partial\Omega_d \subset \partial\Omega$ and $\partial\Omega_n \subset \partial\Omega$, let consider the following conditions:

$$u = g \text{ on } \partial\Omega_d,\ g \in L^2(\partial\Omega_d) \tag{3.5}$$

$$\partial_n u = h \text{ on } \partial\Omega_n,\ h \in L^2(\partial\Omega_n) \tag{3.6}$$

To manage such conditions, we introduce: (i) extra degree of freedoms on boundary faces, (ii) constraints on the bilinear form or (iii) extra terms in the linear form. These contraints and terms lead to add or remove some equations in the matrix and to add extra terms in the right hand side of the linear system.

In our DSEL, the keyword `trace(u)` enables us to recover degrees of freedom on mesh elements, and **on**`(.,.)` enables us to add contraints on groups of mesh elements. For example, with the hybrid method the boundary conditions 3.5 and 3.6 are expressed with the expressions of listing 3.21

Listing 3.21: boundary conditions management

```
BilinearForm ah = integrate(allCells(Th), dot(K*grad(u),grad(v)) ;
LinearForm bh = integrate(allCells(Th),f*v) ;

//Dirichlet condition on ∂Ω_d
ah += on(boundaryFaces(Th,"dirichlet"),trace(u)=g) ;
```

```
//Neumann condition on ∂Ω_n
bh += integrate(boundaryFaces(Th,"neumann"),h*trace(v)) ;
```

3.4 Extensions for multiscale methods

In this section we extend our computational framework to the multiscale methods described in §2.5. We first complete the DSEL front end by introducing new C++ concepts to handle meshes with a fine and a coarse level, function spaces and their basis functions. We extend then the DSEL to refer to restriction-interpolator operators, to design lowest order methods on a coarse level and to interpolate coarse solution on the fine level.

3.4.1 Multiscale mesh

The `MultiscaleMesh` concept extends the `Mesh` concept defined in §2.5.2. It grants access to a coarse mesh built from a fine one. The free functions listed in table 3.8 are provided to give access to the mapping between coarse and fine mesh elements.

3.4.2 Basis function

The `BasisFunction` concept allows to implement the mathematical basis functions ϕ_{σ^c} described in §2.5.3. Associated to a coarse item of the coarse mesh, it defines `DomainType` modeling $\Omega_{\sigma^c} = \text{supp}\{\phi_{\sigma^c}\}$ the support of the basis function. The functions `getDomain()` and `getX()` give access to the basis function support and to the solution of the PDE. A `compute()` function is provided to solve the local PDE problem of which the basis function is a solution.

Listing 3.22: BasisFunctionSpace concept

```
class BasisFunction
{
public:
  typedef ...       DomainType;
  typedef ...       ItemType;
  typedef ...       FunctionType;
  void init();
  void prepare();
  void start();
  void compute();

  DomainType const& getDomain() const;
  FunctionType const& getX() const;
}
```

When basis functions are associated to coarse faces, the basis function support is defined with the back and the front cell of the face. The concept of `HalfBasisFunction` represents the association of a basis function and the back or the front cell of its support.

The `template<typename BasisFunction> GradValueT` gives access to the gradient value of a basis function $\nabla\phi_{\sigma^c}$ described in §2.5.3.

A linear algebra is provided to easily compute expressions like $\lambda\nabla\phi_{\sigma_1^c}+\mu\nabla\phi_{\sigma_2^c}$ and $\nu\nabla\phi_{\sigma_1^c}\cdot\nabla\phi_{\sigma_2^c}$ used in the assembly phase of the coarse linear systems.

3.4.3 Multiscale functional space

The `MultiscaleFunctionSpace` concept represents the mathematical concept of multiscale function space V^{hms} described in 2.5.3. It extends the `FunctionSpace` described in §3.2 and defines the

subtypes `FunctionType`, `TrialFunctionType` and `TestFunctionType`. It is based on a multiscale mesh defining a coarse and a fine mesh, and on a collection of basis functions associated to a collection of items of the coarse mesh which can be computed and updated. Moreover, it defines the `DofType` representing DOFs and implements the **eval()** and **grad()** functions of for the coarse basis functions. The **grad()** function returns a linear combination of `GradvalueT<BasisFunction>` that enables to compute the terms like $\int_{\tau^c} \nu \nabla \phi_1 \cdot \nabla \phi_2$ and $\int_{\sigma^c} \nu \nabla \phi \cdot \mathrm{n}_\sigma$.

The basis function of the Hybrid Multiscale Method described in §2.5.3 is implemented with the `HBasisFunction` class cf. listing3.23. Instances of this class are associated to a coarse face and implement the bilinear forms as in listing 3.24

Listing 3.23: BasisFunction concept

```
template<typename MultiScaleDomainT>
class HBasisFunction
{
public:
  typedef enum { Back, Front }                    ePosType;
  typedef typename MultiScaleDomainT::FineType    DomainType;
  typedef typename MultiScaleDomainT::FaceType    ItemType;

  HBasisFunction(ItemType const& face);

  HalfBasisFunction* halfBasis(ePosType pos);
}
```

Listing 3.24: BasisFunction linear and bilinear form

```
{
  CoarseFace cface = /*...*/;
  CellRealVariable& k = /*...*/;
  CellRealVariable& w = computeWeight(k);
  auto Th = build(cface);
  auto space = newHybridSpace(Th);
  auto u = space->test("U");
  auto v = space->test("V");
  BilinearForm ah = integrate( allCells(Th),k*dot(grad(u),grad(v)) );
  LinearForm bh = integrate( allCells(Th),w*v);
}
```

The template class `MultiScaleFunctionalSpace` (listing 3.25) implements the hybrid multiscale functional spaces V^{hms} described in §2.5.3. This class instantiates the basis function on the coarse faces of the coarse mesh. It implements the three main functions:

- `void start(`κ`,IMultiSystemSolver* solver)` to compute all the basis functions for a given permeability tensor κ and a algebraic layer to solve efficiently collections of independent linear systems;
- **eval**`(Cell const& cell,Integer iface)` returning for $\tau \in \mathcal{T}_H, \sigma \in \partial\tau$ the linear combination
$$\mathtt{l}(\tau,\sigma) = \frac{1}{|\sigma|}\int_\sigma u = u_\tau + \sum_{\sigma' \in \partial\tau} v_{\sigma'} \int_\sigma \phi_{\sigma'}$$
- **grad**`(Cell const& cell)` returning for $\tau \in \mathcal{T}_H$ the linear combination
$$\mathtt{l}(\tau) = \sum_{\sigma \in \partial\tau} v_\sigma \nabla \phi_\sigma$$

.

During the basis functions computation, independent algebraic operations are delegated to an algebraic layer that can perform them efficiently with respect to the hardware configuration. Such layer is described in details in §4.3.3.

During the assembly phase, the evaluation of $\int_\tau \kappa \nabla u \cdot \nabla v$ leads to evaluate the expression **dot(u.grad(cell),v.grad(cell))** which is based on the evaluation of the scalar product of two gradient linear combinations. This evaluation needs for $\sigma_1 \in \mathcal{F}_\tau^c$ and $\sigma_2 \in \mathcal{F}_\tau^c$, the evaluation of $\int_\tau \kappa \nabla \phi_{\sigma_1} \cdot \nabla \phi_{\sigma_2}$ which is implemented as in listing 3.26.

Listing 3.25: MultiscaleFunctionSpace concept

```
//! Define a class representing V^hms
template<typename MultiScaleDomainT,typename BasisFunctionT>
class MultiscaleFunctionSpace
{
public:
  typedef Real                                        ValueType;
  typedef DualNode                                    DofType;
  typedef GradValueT<BasisfunctionT>                  GradValueType;
  typedef LinearCombT<DofType,ValueType>              LinearCombType;
  typedef LinearCombT<DofType,GradValueType>          GradLinearCombType;

  //! compute basis functions
  void start(κ,IMultiSystemSolver* solver);

  LinearCombType eval(Cell const& cell, Integer iface);

  GradLinearCombType eval(Cell const& cell);
}
```

Listing 3.26: basis function scalar product algorithm

```
{
  //! Algorithm to compute ∫_τ κ∇φ1 · ∇φ2
  Real scaMul(HBasisFunction const& basis1,
              HBasisFunction const& basis2,
              VariableCellReal const& k,
              ValueType const& factor) const
  {
    typedef HBasisFunction::DomainType DomainType;
    auto Th = basis1.getDomain();
    auto cells = cells(basis1,basis2,Th);
    if(!cells.empty()) return 0.;
    auto x1 = basis1.getX();
    auto x2 = basis2.getX();
    fvdsel::EvalIntegrateContext<DomainType> ctx(Th);
    return factor*fvdsel::eval(integrate(cells,k*dot(grad(x1),grad(x2))),ctx);
  }
}
```

3.4.4 Extension of the DSEL

Our DSEL has been extended with the new keyword `downscale(.,.)` that enables us to create expressions that can be evaluated and interpolated from the coarse to the fine level following the procedure described in §2.5.4. The `DownscaleEvalContext` object enables us to choose a `DiscreteVariable` on the fine mesh which will be filled with the solution of the interpolation of the coarse expression. Finally, in Listing 3.4.4, we have an illustration of how to implement the hybrid multiscale method described in §2.5.3.

```
MultiscaleMeshType Th ;
Solver coarse_solver = /*...*/;
MultiSystemSolver basis_solver = /*...*/;
Matrix matrix(coarse_solver); //! coarse matrix
Vector rhs(coarse_solver);    //! coarse right hand side

//COARSE PROBLEM DEFINITION
//! create V^hms
auto Uh = newHMSSpace(Th);

//! compute basis functions φ_σ^c
// with a multi-system linear solver layer
Uh->start(κ, basis_solver);

/*...*/
auto u = Uh->trial() ;
auto v = Uh->test() ;
BilinearForm ah =
   integrate( allCells(Th), dot(grad(u),grad(v)) ) +
   integrate( allFaces(Th), -jump(u)*dot(N(Th),avr(grad(v)))
                            -dot(N(Th),avr(grad(u)))*jump(v)
                            +η/H(Th)*jump(u)*jump(v));
ah += on(boundaryFaces(Th), u=ud ) ; //! dirichlet condition
LinearComputeContext lctx(matrix,rhs) ;
fvdsel::eval(ah,lctx);
coarse_solver.solve(matrix,rhs) ;

//FINE PROBLEM SOLUTION
FaceRealVariable& fine_velocity = ... ;
DownScaleEvalContext dctx(fine_velocity) ;
fvdsel::eval(downscale(allCells(Th),flux(u)),dctx) ;
```

3.5 Numerical results

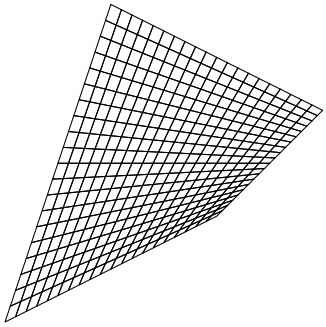
The performance of the DSEL-based implementation of lowest-order methods discussed in Chapter 3 is compared with

- `Feel++`, an open source FE library whose main developer is one of the authors [87]. When possible, `Feel++` is used for comparison with more standard FE methods both in terms of accuracy and performance. The DSEL implemented in `Feel++` has profoundly inspired the present work;
- `fvC++`, an `stl`-based implementation of the back-end discussed in §3.1 used in [49, 52, 53]. The matrix assembly in `fvC++` closely resembles Listing 3.7. No language facility is offered in this case.

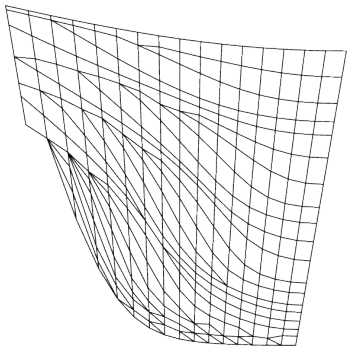
The three codes are compiled with the `gcc` 4.5 compiler with the following compile options:

```
-03 -fno-builtin
-mfpmath=sse -msse -msse2 -msse3 -mssse3 -msse4.1 -msse4.2
-fno-check-new -g -Wall -std=c++0x
--param -max-inline-recursive-depth=32
--param max-inline-insns-single=2000
```

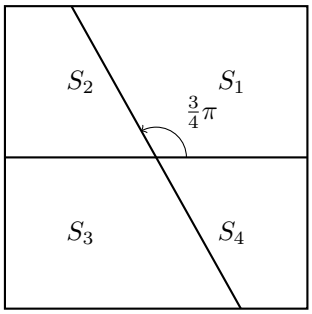
The benchmark test cases are run on a work station with a quad-core Intel Xeon processor GenuineIntel W3530, 2.80GHz, 8MB for cach size.



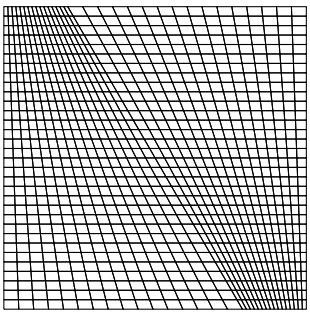
(a) Skewed quadrangular mesh



(b) Stratigraphic mesh. The actual aspect ratio is 10:1



(c) Partition of the domain Ω for the test cases of §3.5.4



(d) Tilted quadrangular mesh for the test cases of §3.5.4

Figure 3.6: Elements of the mesh families used in the benchmark

3.5.1 Meshes

In our numerical tests we consider the following families of h-refined meshes: (i) the skewed quadrangular mesh family of Figure 3.6(a) generated using Gmsh [67] is used for the benchmarks of Sections 3.5.4 and 3.5.5; (ii) the stratigraphic mesh family of Figure 3.6(b) representing a geological basin is used for the benchmark of §3.5.4. This mesh family mixes triangular and quadrangular elements. The actual aspect ratio of the mesh is 10:1, resulting in elongated elements and sharp angles. (iii) the tilted quadrangular mesh family of Figure 3.6(d) is used for the benchmark of Sect. 3.5.4. This mesh family is consistent with the partition of the domain depicted in Figure 3.6(c).

3.5.2 Solvers

The linear systems are solved using the PETSc library. For the diffusion benchmark of §3.5.4, we use the BICGStab solver preconditioned by the euclid ILU(2) preconditioner, with relative tolerance set to 10^{-13}. For the Stokes benchmark of §3.5.5, we use the GMRes solver with a ILU(3) preconditioner and a relative tolerance of 10^{-13}. The constant null space constraint option is activated to solve the system, and the resulting discrete pressure is scaled to ensure that the zero-mean constraint (3.10d) is satisfied. Note that our objective is not to test the solvers but rather compare for a given solution strategy the behavior of the various methods exposed in Chapter 2 as well as more conventional FE methods.

3.5.3 Benchmarks metrics

The benchmarks proposed in this section monitor various metrics:

(i) *Accuracy.* The accuracy of the methods is evaluated in terms of the L^2- and of discrete energy-norms of the error. For the methods of Chapter 2, the L^2-norm of the error is evaluated using the cell center as a quadrature node, i.e.,

$$\|u - u_h\|_{L^2(\Omega)} \approx \left(\sum_{T \in \mathcal{T}_h} |T| (u(\mathbf{x}_T) - u_T)^2 \right)^{\frac{1}{2}}.$$

The actual definition of the discrete energy-norm is both problem and method dependent. Further details are provided for each test case. The convergence order of a method is classically expressed relating the error to the meshsize h.

(ii) *Memory consumption.* When comparing methods featuring different number of unknowns and stencils, a fairer comparison in terms of system size and memory consumption is obtained relating the error to the number of DOFs (N_{DOF}) and to the number of nonzero entries of the corresponding linear system (N_{nz}).

(iii) *Performance.* The last set of parameters is meant to evaluate the CPU cost for each method and implementation. To provide a detailed picture of the different stages and estimate the overhead associated to the DSEL, we separately evaluate:

- t_{init}, the time to build the discrete space;
- t_{ass}, the time to fill the linear systems (local/global assembly). When DSEL-based implementations are considered, this stage carries the additional cost of evaluating the expression tree for bilinear and linear forms;
- t_{solve}, the time to solve the linear system.

An important remark is that, in the context of nonlinear problems on fixed meshes, t_{init} often corresponds to precomputation stages, while t_{ass} contributes to each iteration.

3.5.4 Pure Diffusion benchmarck

We consider the standard Poisson problem:

$$\begin{aligned} -\triangle u &= 0 \text{ in } \Omega \subset \mathbb{R}^3 \\ u &= g \text{ on } \partial\Omega \end{aligned} \tag{3.7}$$

The continuous weak formulation reads: Find $u \in H_0^1(\Omega)$ such that

$$a(u, v) = 0 \quad \forall v \in H_0^1(\Omega),$$

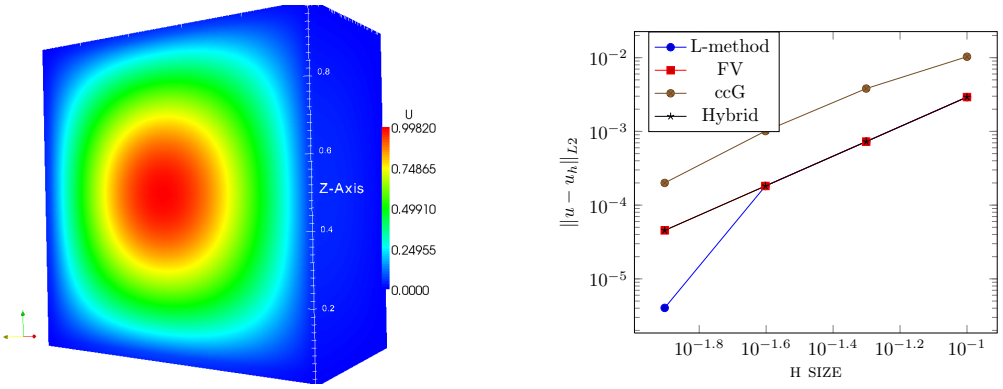
with

$$a(u, v) \overset{\text{def}}{=} \int_\Omega \nabla u \cdot \nabla v.$$

The discrete formulations of the problem with the G-method, the ccG-method and the Hybrid-method defined in Chapter 2 are represented by the definition of the bilinear forms a_h^{g}, a_h^{ccg}, a_h^{hyb} and the linear form b_h. We can compare them to their programming counterpart in listings 3.27,3.28 and 3.29

Listing 3.27: C++ implementation of a_h^{g} and b_h

```
MeshType Th; // declare T_h
auto Vh = newP0Space(Th);
auto Uh =  newGSpace(Th);
auto u = Uh->trial("U");
auto v = Vh->test("V");
BilinearForm ah_g =
 integrate(allFaces(Th),dot(N(),avg(grad(u)))*jump(v));
LinearForm bh =
  integrate( allCells(Th), f*v );
```



(a) 3D view (b) convergence curves

Figure 3.7: Diffusion problem

Listing 3.28: C++ implementation of a_h^{ccg}

```
MeshType Th; // declare T_h
auto Uh = newCCGSpace(Th);
auto u = Uh->trial("U");
auto v = Uh->test("V");
auto lambda = eta*gamma/H();
BilinearForm ah_ccg =
  integrate(allCells(Th),dot(grad(u),grad(v))) +
  integrate(allFaces(Th),-jump(u)*dot(N(),avg(grad(v)))
                         -dot(N(),avg(grad(u)))*jump(v)
                         +lambda*jump(u)*jump(v);
```

Listing 3.29: C++ implementation of a_h^{hyb}

```
MeshType Th; // declare T_h
auto Uh = newHybridSpace(Th);
auto u = Uh->trial("U");
auto v = Uh->test("V");
BilinearForm ah_hyb =
 integrate(allFaces(Th),dot(grad(u),grad(v));
```

3D results

We consider the analytical solution $u(x, y, z) = sin(\pi x)sin(\pi y)sin(\pi z)$ of the diffusion problem 3.7 on the square domain $\Omega = [0, 1]^3$ with $f(x, y, z) = 3\pi^2 sin(\pi x)sin(\pi y)sin(\pi z)$.

Table 3.9, 3.10 and 3.11 list the errors in the L^2 and L norms of respectively the G method, the ccG method and the hybrid method.

In Figure 3.7, we compare the convergence error of the G method, the ccG method, the SUSHI method and a standard hand written L Scheme FV method.

In the tables 3.12, 3.13, 3.14 and 3.15, we compare the performance of each methods.

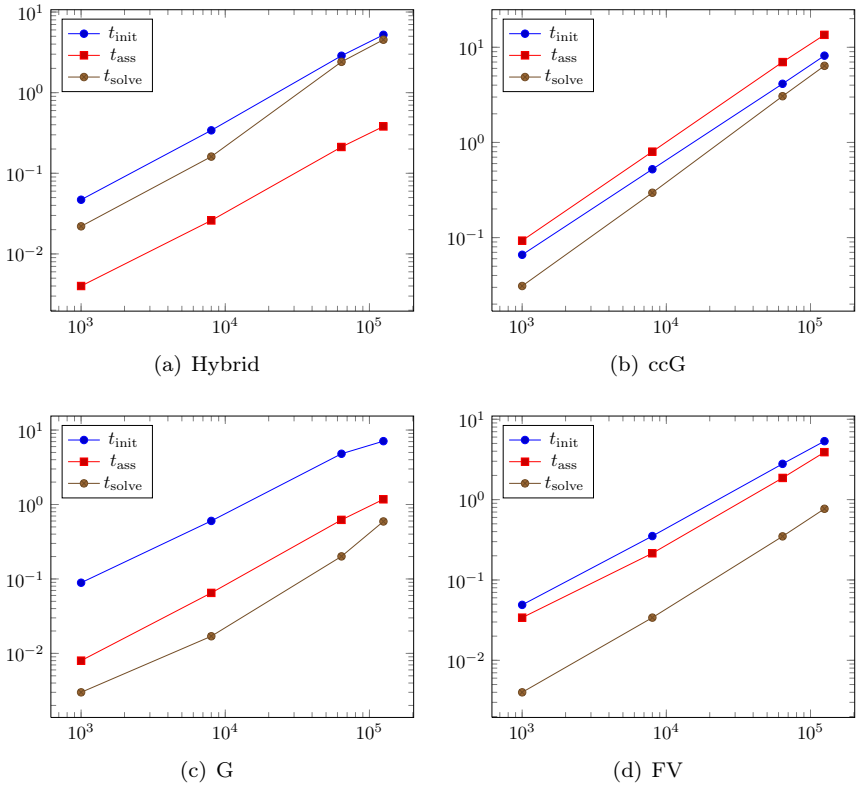
The analysis of these results shows that the G-method is comparable to the hand written FV method and the language implementation does not contribute to extra cost. The G-method and the Hybrid-method have equivalent convergence order. A closer look to the N_{nz} column shows that the ccG method requires much more nonzero entries for the linear system than the G-method

Table 3.6: DSEL keywords

keyword	meaning
integrate(.,.)	$\int(.)$ integration of expression
dot(.,.)	$(.\cdot.)$ vector inner product
jump(.)	$[\![.]\!]$ jump accross a face
avg(.)	$\{.\}$ average accross a face

Table 3.7: Proto standard tags and meta-functions

operator	narity	tag	meta-function
+	2	proto::tag::plus	proto::plus<.,.>
-	2	proto::tag::minus	proto::minus<.,.>
*	2	proto::tag::**mult**	proto::**mult**<.,.>
/	2	proto::tag::**div**	proto::**div**<.,.>



(a) Hybrid (b) ccG

(c) G (d) FV

Figure 3.8: time vs. N_{DOF}

`fineElements(<mesh>,<cell>)`	fine elements of a coarse cell
`fineElements(<mesh>,<face>)`	fine elements of a coarse face
`backCell(<mesh>,<face>)`	back cell a coarse face,
`frontCell(<mesh>,<face>)`	front cell of a coarse face,
`boundaryCell(<mesh>,<face>)`	boundary cell of a boundary coarse face,

Table 3.8: Multiscale mesh element extraction tools

Table 3.9: Diffusion test case: G method

$\text{card}(\mathcal{T}_h)$	h	$\|u-u_h\|_L$	order	$\|u-u_h\|_{L^2(\Omega)}$	order
1000	$1.00\ 10^{-1}$	$1.58\ 10^{-2}$		$2.92\ 10^{-3}$	
8000	$5.00\ 10^{-2}$	$3.96\ 10^{-3}$	2.	$7.28\ 10^{-4}$	2.
64000	$2.50\ 10^{-2}$	$9.89\ 10^{-4}$	2.	$1.82\ 10^{-4}$	2.
125000	$2.00\ 10^{-2}$	$6.32\ 10^{-4}$	2.	$1.16\ 10^{-4}$	2.

and the hybrid-method, and we can see the effect on the cost of the linear system building phase which is more important for the ccG method than for the G-method.

The inspection of the columns t_{start}/t_{ref} and t_{build}/t_{ref} shows that the implementation remains scalable with respect to the size of the problem.

2D results

Our second benchmark is based on the following exact solution for the diffusion problem (2.4):

$$u(\mathbf{x}) = \sin(\pi x_1)\cos(\pi x_2), \qquad \kappa = \mathbb{1}_d$$

with $\mathbb{1}_d$ identity matrix in $\mathbb{R}^{d,d}$. The right-hand side f is inferred from the exact solution, and Dirichlet boundary conditions are enforced on $\partial\Omega$. The problem is solved on the skewed mesh family depicted in Figure 3.6(a). We compare the following methods: (i) the DSEL and `fvC++` implementations of the ccG method (2.13). The DSEL implementation is provided in Listings 3.1; (ii) the DSEL implementation of the SUSHI method with face unknowns (2.15) provided in Listing 3.12; (iii) the `Feel++` implementation of the first-order Rannacher–Turek elements $\mathbb{RT}^1$; (iv) the `Feel++` implementation of $\mathbb{Q}^k$ elements with $k \in \{1, 2\}$. Since the solution is smooth, the $\mathbb{Q}^2$ element is expected to yield better performance. In real-life applications, however, the regularity of the solution is limited by the heterogeneity of the diffusion coefficient; see [55] and references therein for a discussion.

The accuracy and memory consumption analysis is provided in Figure 3.10. The discrete H^1-norm coincides with the natural coercivity norm for the method; see [53, 65] for further details on the SUSHI and ccG methods. As expected, the higher-order method $\mathbb{Q}^2$ elements yields better

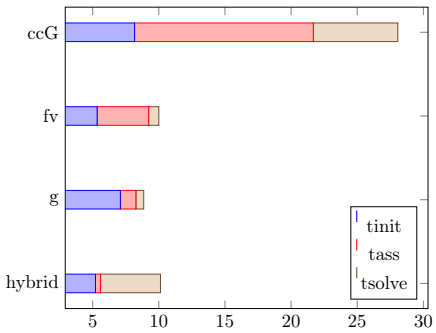


Figure 3.9: time vs. N_{DOF},$h = 0.02$

Table 3.10: Diffusion test case: ccG method

card($\mathcal{T}_h$)	h	$\|u-u_h\|_L$	order	$\|u-u_h\|_{L^2(\Omega)}$	order
1000	$1.00\ 10^{-1}$	$3.1474\ 10^{-2}$		$5.3866\ 10^{-3}$	
8000	$5.00\ 10^{-2}$	$7.8977\ 10^{-3}$	1.99	$1.4257\ 10^{-3}$	1.92
64000	$2.50\ 10^{-2}$	$1.9763\ 10^{-3}$	2.	$3.6157\ 10^{-4}$	1.95
125000	$2.00\ 10^{-2}$	$1.2649\ 10^{-3}$	2.	$2.3180\ 10^{-4}$	1.95

Table 3.11: Diffusion test case: Hybrid method

card($\mathcal{T}_h$)	h	$\|u-u_h\|_L$	order	$\|u-u_h\|_{L^2(\Omega)}$	order
1000	$1.00\ 10^{-1}$	$1.58\ 10^{-2}$		$2.92\ 10^{-3}$	
8000	$5.00\ 10^{-2}$	$3.95\ 10^{-3}$	2.	$7.28\ 10^{-4}$	2.01
64000	$2.50\ 10^{-2}$	$9.87\ 10^{-4}$	2.	$1.82\ 10^{-4}$	2.
125000	$2.00\ 10^{-2}$	$6.32\ 10^{-4}$	2.	$1.16\ 10^{-4}$	2.

Table 3.12: Diffusion test case: G-method performance results

card($\mathcal{T}_h$)	N_{it}	N_{nz}	t_{start}	t_{def}	t_{build}	t_{solve}	t_{ref}	$\frac{t_{start}}{t_{ref}}$	$\frac{t_{def}}{t_{ref}}$	$\frac{t_{build}}{t_{ref}}$
1000	4	16120	$8.89\ 10^{-2}$	$1.19\ 10^{-2}$	$7.99\ 10^{-3}$	$3.00\ 10^{-3}$	$7.50\ 10^{-4}$	118.	16.0	10.6
8000	8	140240	$6.01\ 10^{-1}$	$1.03\ 10^{-1}$	$6.49\ 10^{-2}$	$1.69\ 10^{-2}$	2.1210^{-3}	283.	48.9	30.5
64000	14	1168480	4.80	$8.47\ 10^{-1}$	$6.21\ 10^{-1}$	$2.00\ 10^{-1}$	$1.44\ 10^{-2}$	334.	59.0	43.3
125000	25	2300600	7.09	1.71	1.17	$5.91\ 10^{-1}$	$2.37\ 10^{-2}$	299.	72.3	49.5

Table 3.13: Diffusion test case: ccG-method performance results

card($\mathcal{T}_h$)	N_{it}	N_{nz}	t_{start}	t_{def}	t_{build}	t_{solve}	t_{ref}	$\frac{t_{start}}{t_{ref}}$	$\frac{t_{def}}{t_{ref}}$	$\frac{t_{build}}{t_{ref}}$
1000	3	117642	$6.59\ 10^{-2}$	$3.54\ 10^{-1}$	$9.29\ 10^{-2}$	$3.09\ 10^{-2}$	$1.03\ 10^{-2}$	6.39	34.36	9
8000	5	1145300	$5.22\ 10^{-1}$	3.46	$8.00\ 10^{-1}$	$2.95\ 10^{-1}$	$5.92\ 10^{-2}$	8.83	58.6	13.5
64000	8	10114802	4.13	2.98e1	6.99	3.06	$3.83\ 10^{-1}$	10.8	78.08	18.27
125000	10	20017250	8.16	6.09e1	1.35e1	6.38	$6.39\ 10^{-1}$	12.7	95.3	21.17

Table 3.14: Diffusion test case: Hybrid method performance results

card($\mathcal{T}_h$)	N_{it}	N_{nz}	t_{start}	t_{def}	t_{build}	t_{solve}	t_{ref}	$\frac{t_{start}}{t_{ref}}$	$\frac{t_{def}}{t_{ref}}$	$\frac{t_{build}}{t_{ref}}$
1000	7	16120	$4.69\ 10^{-2}$	$1.09\ 10^{-2}$	$4.00\ 10^{-3}$	$2.19\ 10^{-2}$	$3.14\ 10^{-3}$	14.9	3.5	1.27
8000	17	140240	$3.40\ 10^{-1}$	$1.18\ 10^{-1}$	$2.59\ 10^{-2}$	$1.60\ 10^{-1}$	$9.47\ 10^{-3}$	36.0	12.5	2.75
64000	33	1168480	2.86	1.11	$2.11\ 10^{-1}$	2.41	$7.30\ 10^{-2}$	39.2	15.2	2.9
125000	50	5563700	5.21	2.05	$3.80\ 10^{-1}$	4.53	$9.06\ 10^{-2}$	57.5	22.62	4.2

Table 3.15: Diffusion test case: standard hand written performance results

card($\mathcal{T}_h$)	N_{it}	N_{nz}	t_{start}	$t_{def+build}$	t_{solve}	t_{ref}	$\frac{t_{start}}{t_{ref}}$	$\frac{t_{build}}{t_{ref}}$
1000	4	16120	$4.89\ 10^{-2}$	$3.39\ 10^{-2}$	$3.998\ 10^{-3}$	$1.00\ 10^{-3}$	49.01	34.01
8000	7	140240	$3.51\ 10^{-1}$	$2.14\ 10^{-1}$	$3.399\ 10^{-2}$	$4.86\ 10^{-3}$	72.47	44.26
64000	13	1168480	2.78	1.86	$3.489\ 10^{-1}$	$2.68\ 10^{-2}$	103.8	69.34
125000	16	9536960	5.33	3.89	$7.688\ 10^{-1}$	$4.81\ 10^{-2}$	111.09	81.02

performance whether or not the error is related to the meshsize h, the number of DOFs N_{DOF}, or the number of nonzero elements in the matrix N_{nz}. It has to be noted, however, that both the SUSHI and the ccG methods exhibit superconvergence in the discrete H^1-norm, thereby providing

a better approximation of the gradient with respect to the first-order element methods.

The CPU cost analysis is provided in Figures 3.11 and 3.12. The cost of each stage of the computation is related to the number of DOFs in Figure 3.11 to check that the expected complexity is achieved. This is the case for all the methods considered. A comparison in terms of absolute computation time is provided in Figure 3.12. Overall, the initialization and assembly steps appear more expensive for the lowest-order methods. The overhead of the DSEL can be estimated by comparing with the `fvC++` implementation of the ccG. Some other remarks can be done: (i) the main interest of the lowest-order methods presented in Chapter 2 is that general meshes can be handled seamlessly. For an example based on a less conventional mesh see §3.5.4. When a classical FE implementation is possible, the approach based on a reference element and a table of DOFs can be expected to overperform the `LinearCombination`-based handling of degrees of freedom; (ii) the FE code `Feel++` is a more mature project, which benefits from some degree of optimization and finer tuning; (iii) even when the overhead of the DSEL is disregarded, the `stl`-based implementation of `LinearCombination` in `fvC++` yields similar performance as the dedicated implementation used in the DSEL version; (iv) The solution times are slightly different between the `fvC++` and DSEL version of lowest-order methods owing to a different degree of optimization in handling the matrix profile. Indeed, in the `fvC++` implementation of `LinearCombinationBuffer`, zero coefficients are expunged when a `LinearCombination` is computed; cf. §3.1.3. As a result, in some circumstances the matrices in `fvC++` are sparser than in the DSEL implementation.

Heterogeneous results

We consider in this section two exact solutions for a heterogeneous medium originally proposed in [27]. The regularity of the solutions is affected by the heterogeneity of the diffusion tensor, and is insufficient to attain the maximum convergence rate for some or all of the considered methods. In the context of geological simulations, this test case models some of the difficulties encountered when faults are present. The domain $\Omega = (0,1)^2$ is partitioned into four areas corresponding to different values of the diffusion coefficient κ as depicted in Figure 3.6(c), and we consider the tilted mesh family of Figure 3.6(d). The permeability coefficient is such that $\kappa|_{S_1} = k_1 \mathbb{1}_d$ and $\kappa|_{\Omega\setminus S_1} = k_2 \mathbb{1}_d$. Using polar coordinates (r,θ) with $\theta = \cos^{-1}(x_1/r)$ and origin at the center of the domain, the first solution is given by

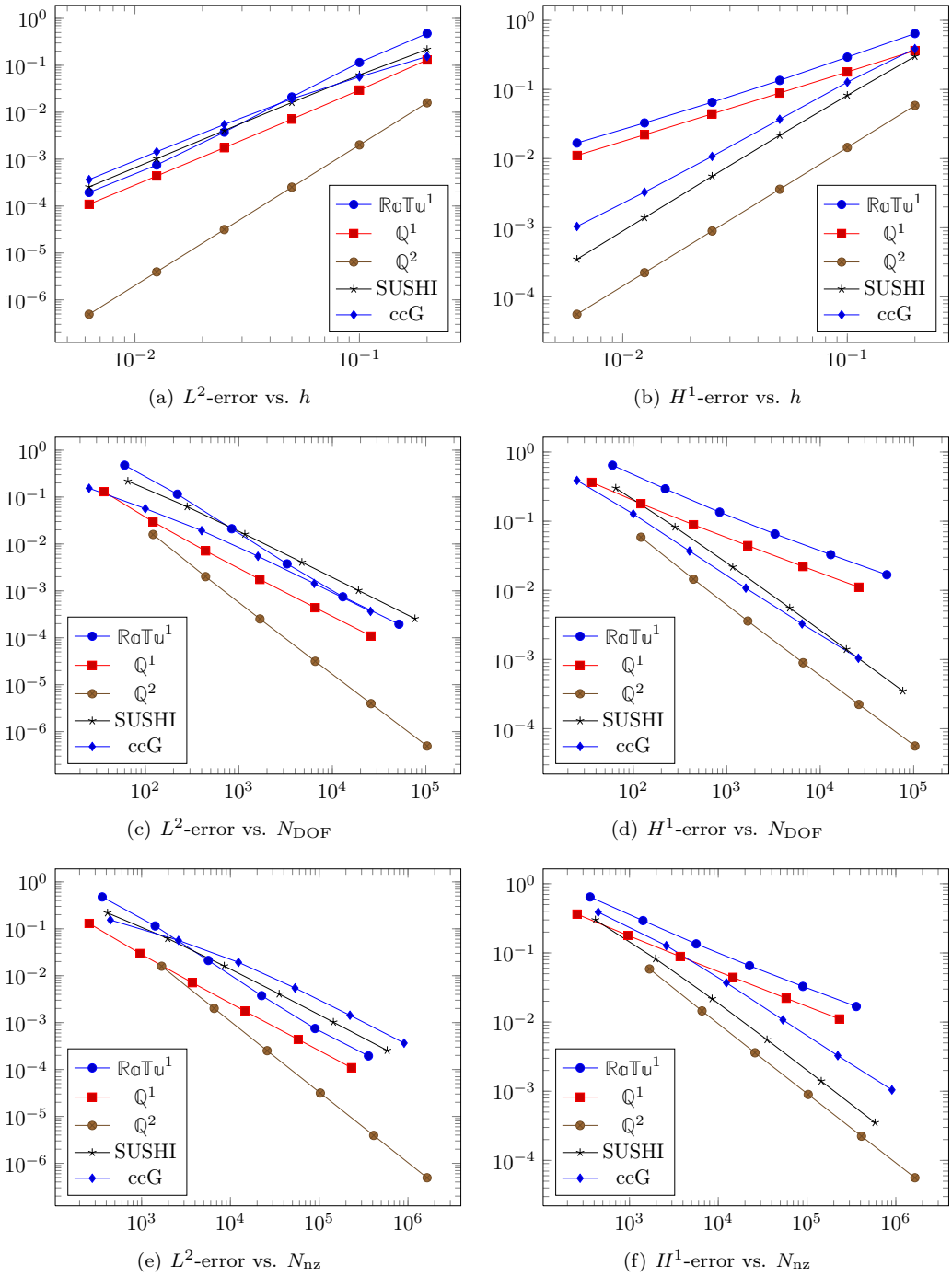
$$u = \begin{cases} r^\alpha \cos\left(\alpha(\theta - \frac{\pi}{3})\right) & \text{if } \theta \in \left[0, \frac{2\pi}{3}\right), \\ r^\alpha \beta \cos\left(\frac{4\pi}{3} - \theta\right) & \text{if } \theta \in \left[\frac{2\pi}{3}, 2\pi\right), \end{cases} \tag{3.8}$$

where $\alpha = \frac{3}{\pi}\tan^{-1}\left(\sqrt{1+\frac{2}{\epsilon}}\right)$, $\beta = \cos\left(\alpha\frac{\pi}{3}\right)/\cos\left(2\alpha\frac{\pi}{3}\right)$ and $\epsilon = k_1/k_2$ is the heterogeneity ratio taken equal to 0.1. It can be proved that $u \in H^{2.29}(\Omega)$. While this solution has sufficient regularity to attain the optimal order of convergence for lowest-order methods, it does not allow to fully exploit $\mathbb{Q}^2$ or higher order elements. The expected behaviour is confirmed by the accuracy analysis of Figure 3.13.

A second solution with less regularity is the following:

$$u = \begin{cases} r^\alpha \sin\left(\alpha(\theta - \frac{\pi}{3})\right) & \text{if } \theta \in \left[0, \frac{2\pi}{3}\right), \\ r^\alpha \beta \sin\left(\alpha(\frac{4\pi}{3} - \theta)\right) & \text{if } \theta \in \left[\frac{2\pi}{3}, 2\pi\right), \end{cases} \tag{3.9}$$

where now $\alpha = \frac{3}{\pi}\tan^{-1}\left(\sqrt{1+2\epsilon}\right)$, $\beta = \left(2\cos(\alpha\frac{\pi}{3})\right)^{-1}$ and, as before, $\epsilon = k_1/k_2 = 0.1$. In this case, $u \in H^{1.79}(\Omega)$, so that even lowest order methods cannot attain the maximum order of convergence. The expected behaviour is confirmed by the accuracy analysis of Figure 3.14. It is interesting to note that the H^1 error norm in the lowest order method and $\mathbb{RTU}^1$ behaves better than the $\mathbb{Q}^k$ methods and conversely the L^2 norm behaves better using the $\mathbb{Q}^k$ methods. This is expected as the lowest order methods are designed to recover better approximations for the gradient.



(a) L^2-error vs. h

(b) H^1-error vs. h

(c) L^2-error vs. N_{DOF}

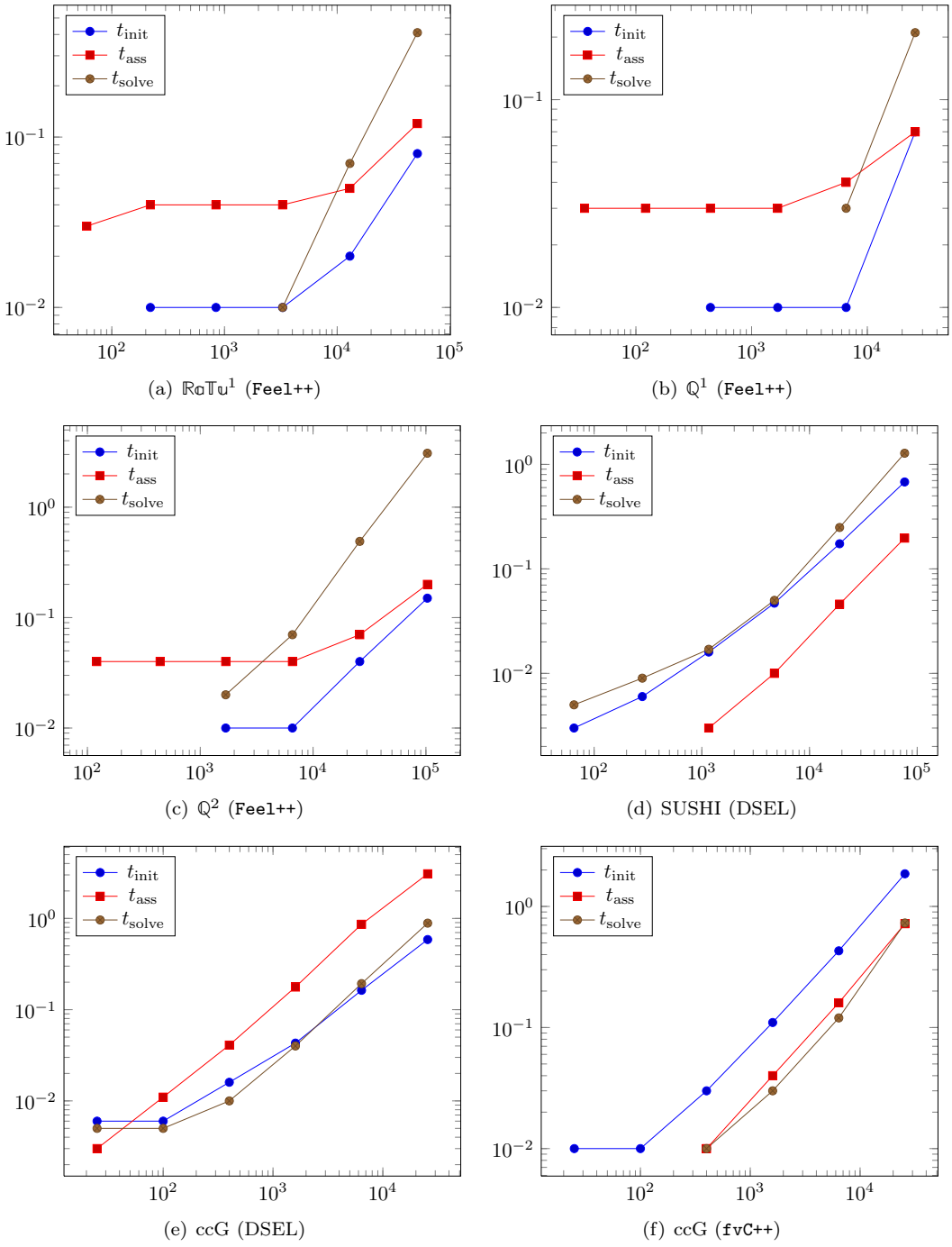
(d) H^1-error vs. N_{DOF}

(e) L^2-error vs. N_{nz}

(f) H^1-error vs. N_{nz}

Figure 3.10: Accuracy and memory consumption analysis for the example of §3.5.4

Remark 3 (Implementation note). The Q^k implementation requires to use a weak formulation to handle the Dirichlet conditions to ensure optimal (or the best possible) convergence. This is due to the localisation of the sharp solution features at the degrees of freedom on the boundary. The weak formulation requires interpolation only at interior edge points. $\mathbb{RT}^1$ can use strong Dirichlet treatment — the degrees of freedom are located at the middle of the edges — as well as the weak one.



(a) $\mathbb{R}\mathbb{o}\mathbb{T}\mathbb{u}^1$ (`Feel++`)

(b) $\mathbb{Q}^1$ (`Feel++`)

(c) $\mathbb{Q}^2$ (`Feel++`)

(d) SUSHI (DSEL)

(e) ccG (DSEL)

(f) ccG (`fvC++`)

Figure 3.11: Performance analysis for the example of §3.5.4

A problem in basin modeling

The last problem is based on the basin mesh family depicted in Figure 3.6(b) which contains both triangular and quadrangular elements. Handling this kind of mesh usually requires some specific modifications in finite element codes, since two reference finite elements exists. A key advantage of the lowest-order methods considered in the present work is that their construction remains

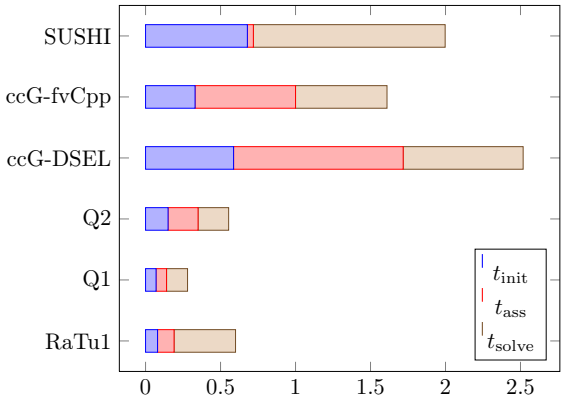


Figure 3.12: Comparison of different methods and implementation for the test case of §3.5.4 (time vs. N_{DOF}, $h = 0.00625$)

unchanged for elements of different shape. We consider the anisotropic test case of [30],

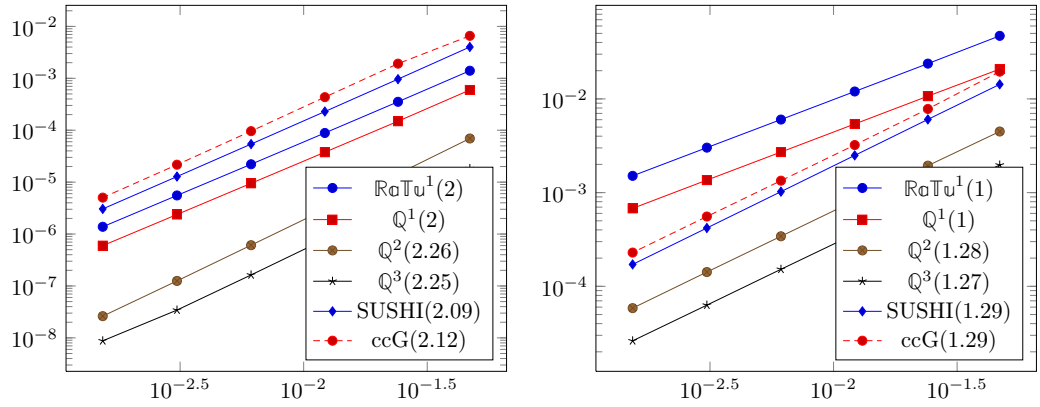
$$u(\mathbf{x}) = \sin(\pi x_1)\sin(\pi x_2), \qquad \kappa = \begin{bmatrix} \epsilon & 0 \\ 0 & 1 \end{bmatrix},$$

with suitable right-hand side f and Dirichlet boundary conditions on $\partial\Omega$. The anisotropy ratio ϵ is taken equal to 0.1. We compare the following discretizations: (i) the G-method (2.8) whose DSEL implementation is provided Listing 3.30; (ii) the ccG method (2.13); (iii) the SUSHI method (2.15) with discrete gradient (2.16) expressed in terms of cell unknowns only.

Listing 3.30: Implementation of the G-method (2.8)

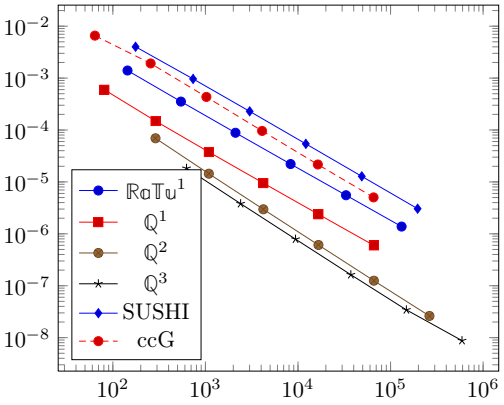
```
Mesh Th(/* ... */);
auto Uh = newGSpace(Th);
auto Vh = newP0Space(Th);
auto uh = *Uh->trial("uh");
auto vh = *Vh->test("vh");
BilinearForm ah = integrate(allFaces(Th),
                            -dot(N(),avg(K*grad(uh)))*jump(vh)
                            );
LinearForm bh = integrate(allCells(Th), f*v);
```

The difficulty in this case is related to both the mesh, which mixes elongated triangular and quadrangular elements, and the anisotropy of the diffusion tensor. The results are presented in Figure 3.15. To facilitate the comparison with the results of [30, Figure 5], the discrete energy norm is defined according to [30, eq. (4.1)] for both the G-method and the SUSHI scheme, while for the ccG method we have used the norm of [53, eq. (3.7)]. While all of the methods have cell centered unknowns only, their stencils differ significantly. It is interesting to remark that, despite its larger stencil, the ccG method outperforms both the G-method and the SUSHI methods in terms of the discrete H^1-norm even when relating the error to the number of nonzero elements in the matrix. On the other hand, the G-method displays good convergence properties in the L^2-norm, but its performance is poor when it comes to the discrete H^1-norm. Finally, the SUSHI method may be a compromise when the memory occupation of the ccG method becomes unacceptable.

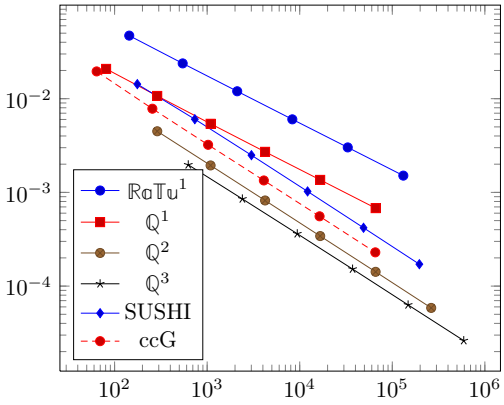


(a) L^2-error vs. h

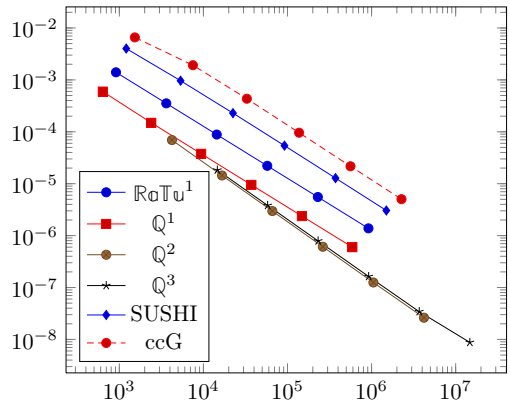
(b) H^1-error vs. h



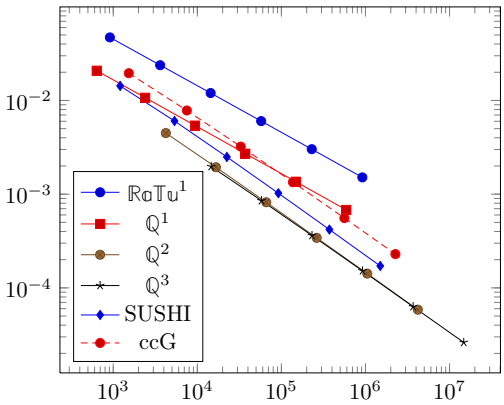
(c) L^2-error vs. nunkw



(d) H^1-error vs. nunkw

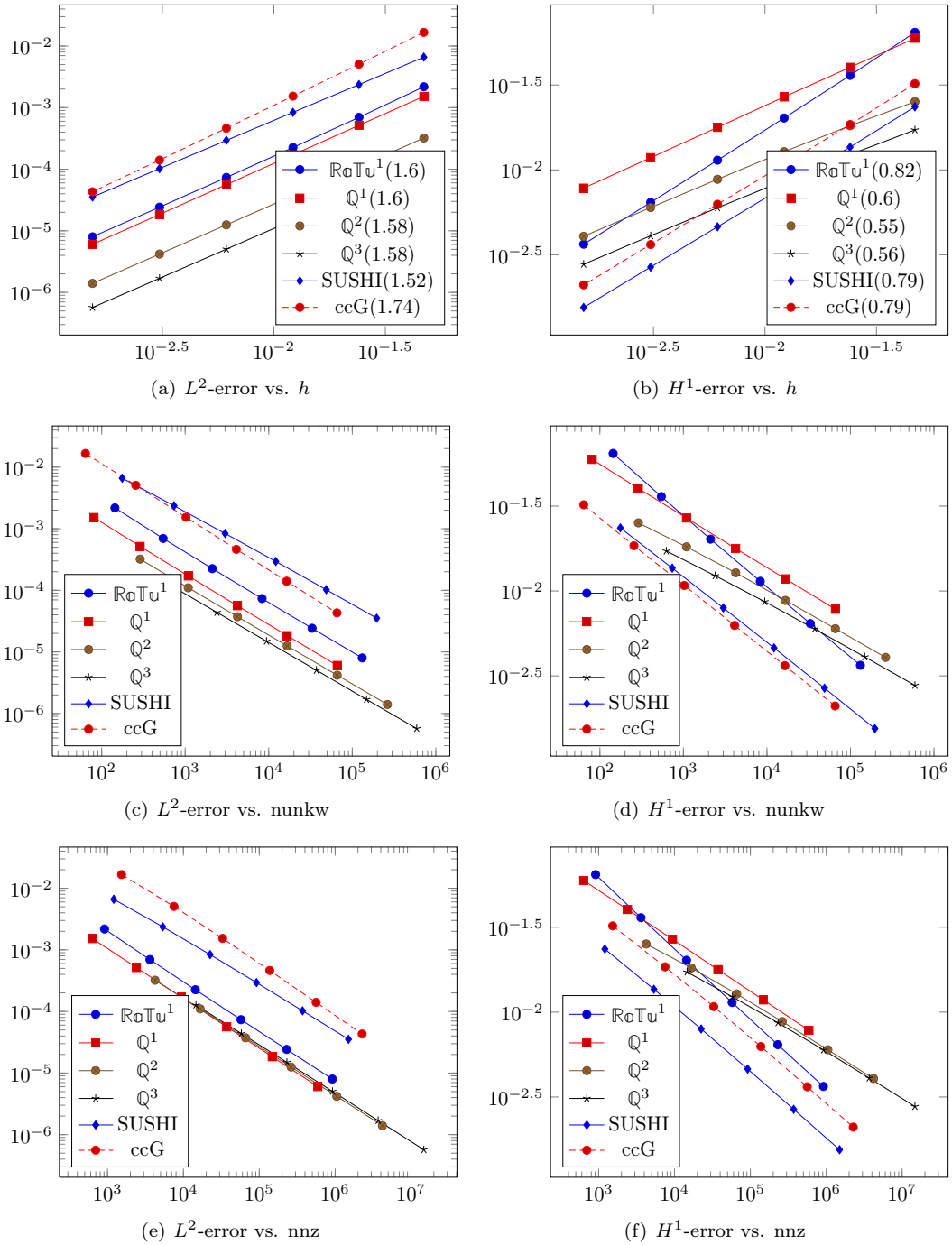


(e) L^2-error vs. nnz



(f) H^1-error vs. nnz

Figure 3.13: Accuracy and memory consumption analysis for the heterogeneous diffusion example (3.8)



(a) L^2-error vs. h

(b) H^1-error vs. h

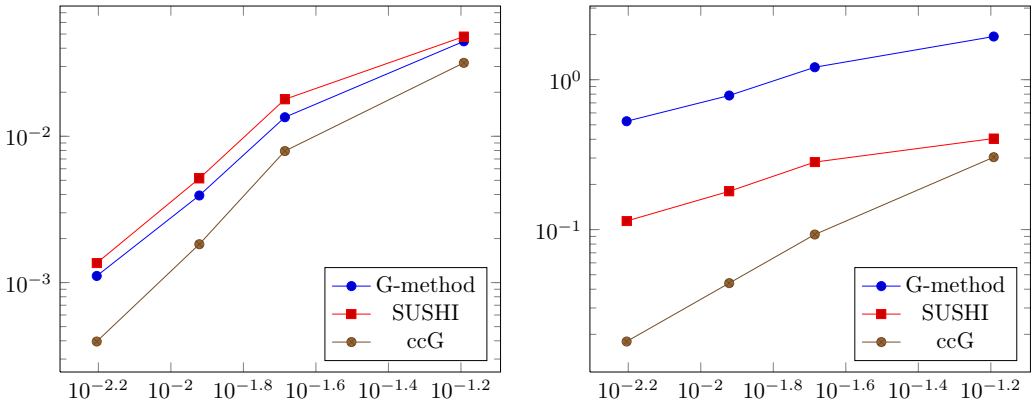
(c) L^2-error vs. nunkw

(d) H^1-error vs. nunkw

(e) L^2-error vs. nnz

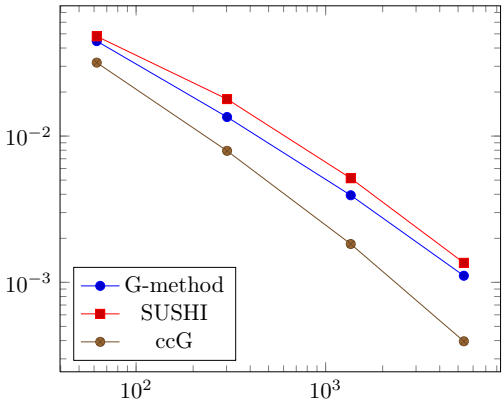
(f) H^1-error vs. nnz

Figure 3.14: Accuracy and memory consumption analysis for the heterogeneous diffusion example (3.9)

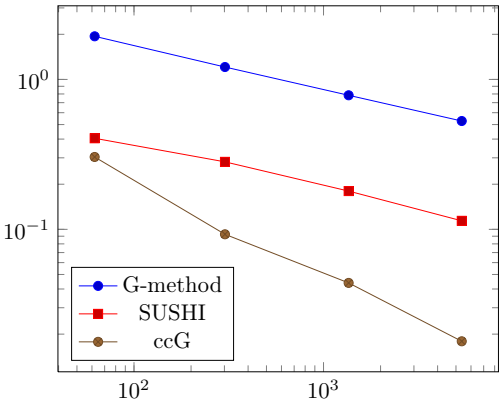


(a) L^2-error vs. h

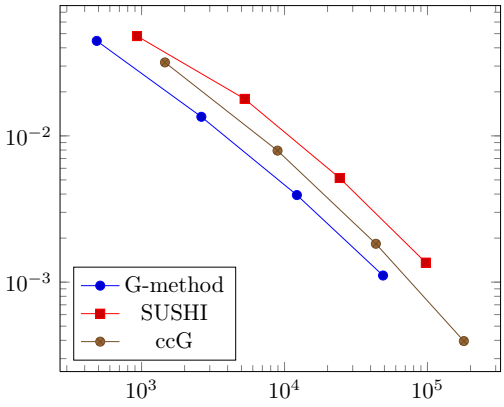
(b) H^1-error vs. h



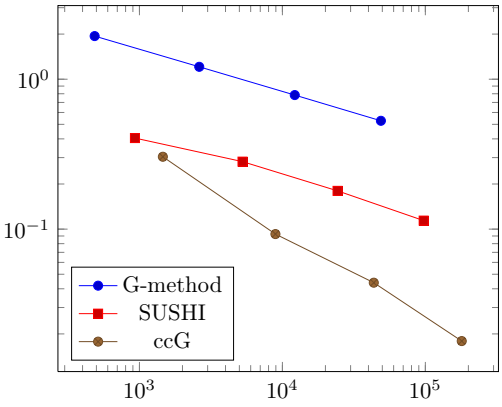
(c) L^2-error vs. N_{DOF}



(d) H^1-error vs. N_{DOF}



(e) L^2-error vs. N_{nz}



(f) H^1-error vs. N_{nz}

Figure 3.15: Accuracy and memory consumption analysis for the example of §3.5.4

3.5.5 Stokes benchmarck

We consider the Stokes problem3.5.5:

$$-\nu\triangle\mathbf{u}+\nabla p=f \qquad \text{in } \Omega, \tag{3.10a}$$
$$\nabla\cdot\mathbf{u}=0 \qquad \text{in } \Omega, \tag{3.10b}$$
$$\mathbf{u}=0 \qquad \text{on } \partial\Omega, \tag{3.10c}$$
$$\int_\Omega p=0, \tag{3.10d}$$

where $u:\Omega\to\mathbb{R}^d$ is the vector-valued velocity field, $p:\Omega\to\mathbb{R}$ is the pressure, and $f:\Omega\to\mathbb{R}^d$ is the forcing term. Equations (3.10a) and (3.10b) express the conservation of momentum and mass respectively. The problem is supplemented by the homogeneous boundary condition (3.10c)

The continuous weak formulation for $g=0$ reads:

Find $(\mathbf{u},p)\in[H_0^1(\Omega)]^d\times L_*(\Omega)$ such that

$$a(\mathbf{u},\mathbf{v})+b(p,\mathbf{v})-b(q,\mathbf{u})=\int_\Omega f\cdot\mathbf{v}\quad\forall(\mathbf{v},q)\in[H_0^1(\Omega)]^d\times L_*(\Omega),$$

with

$$a(\mathbf{u},\mathbf{v})\overset{\text{def}}{=}\int_\Omega\nu\nabla\mathbf{u}{:}\nabla\mathbf{v},\quad b(q,\mathbf{v})\overset{\text{def}}{=}-\int_\Omega\nabla q\cdot\mathbf{v}=\int_\Omega q\nabla\cdot v.$$

Set $c((\mathbf{u},p),(\mathbf{v},q))\overset{\text{def}}{=}a(\mathbf{u},\mathbf{v})+b(p,\mathbf{v})-b(q,\mathbf{u})$.

The discretization of the variational formulation with a ccG method is:

$$\begin{aligned}a_h(u_h,v_h)\overset{\text{def}}{=}\int_\Omega&-\nu\nabla u_h\cdot\nabla v_h\\&+\sum_{F_h\in\Omega_h}\int_{F_h}-\nu[\![u_h]\!](\{\nabla u_h\}\cdot\mathrm{n}_{F_h})-\nu(\{\nabla u_h\}\cdot\mathrm{n}_{F_h})[\![v_h]\!]\\&+\sum_{F_h\in\partial\Omega_h}\int_{F_h}-\nu[\![u_h]\!](\{\nabla v_h\}\cdot\mathrm{n}_{F_h})-\nu(\{\nabla u_h\}\cdot\mathrm{n}_{F_h})[\![v]\!]\end{aligned} \tag{3.11}$$

$$b_h(p_h,v_h)\overset{\text{def}}{=}\sum_{T_h\in\mathcal{T}_h}\int_{T_h}-p_h\nabla\cdot v_h+\sum_{F_h\in\mathcal{F}_h}\int_{F_h}\{p_h\}(\mathrm{n}_{F_h}\cdot[\![v_h]\!]))$$
$$b_h(p_h,v_h)\overset{\text{def}}{=}\sum_{T_h\in\mathcal{T}_h}\int_{T_h}\nabla p_h\cdot v_h-\sum_{F_h\in\mathcal{F}_h^i}\int_{F_h}[\![p_h]\!](\mathrm{n}_{F_h}\cdot\{v_h\})$$

This formulation can be compared to its programming counterpart in listings 3.31.

Listing 3.31: C++ implementation of the Stokes problem

```
MeshType Th ; // declare Th
Real nu, eta ; //declare nu, eta
VariableArray<Real> f ; //declare vectorial source term f
auto Uh = newCCGSpace(Th) ;
auto Ph = newP0Space(Th) ;
auto u = *Uh->trial("U",Th::dim) ;
auto v = *Uh->test("V",Th::dim) ;
auto p = *Ph->trial("P") ;
auto q = *Ph->test("Q") ;
FVDomain::algo::Range<1> range(dim) ;
```

```
FVDomain::algo::Index& _i = range.get<0>() ;
BilinearForm ah =
  integrate( allCells(Th),
             sum(_i)[
                      nu*dot(grad(u(_i)),grad(v(_i)))
                    ]
           ) +
  integrate( internalFaces(Th),
             sum(_i)[  -nu*dot(N(),avr(grad(u(_i)))) * jump(v(_i)) -
                       nu*jump(u(_i))*dot(N(),avr(grad(v(_i)))) +
                       eta/H()*jump(u(_i))*jump(v(_i))
                    ]
           ) ;

BilinearForm bh =
  integrate( allCells(Th), -p*div(v) ) +
  integrate( allFaces(Th), avr(p)*dot(fn,jump(v)) ) ) ;

BilinearForm bth =
  integrate( allCells(Th), div(u)*id(q) ) +
  integrate( allFaces(Th), -dot(N(),jump(u)) * avr(q) ) ;

BilinearForm sh =
  integrate( internalFaces(Th), H()*jump(p)*jump(q) ) ;

LinearForm form1 =
   integrate( allCells(Th),sum(_i)[ f(_i)*v(_i) ] ) ;
```

Results

We consider the following analytical solution of the Stokes problem (3.5.5):

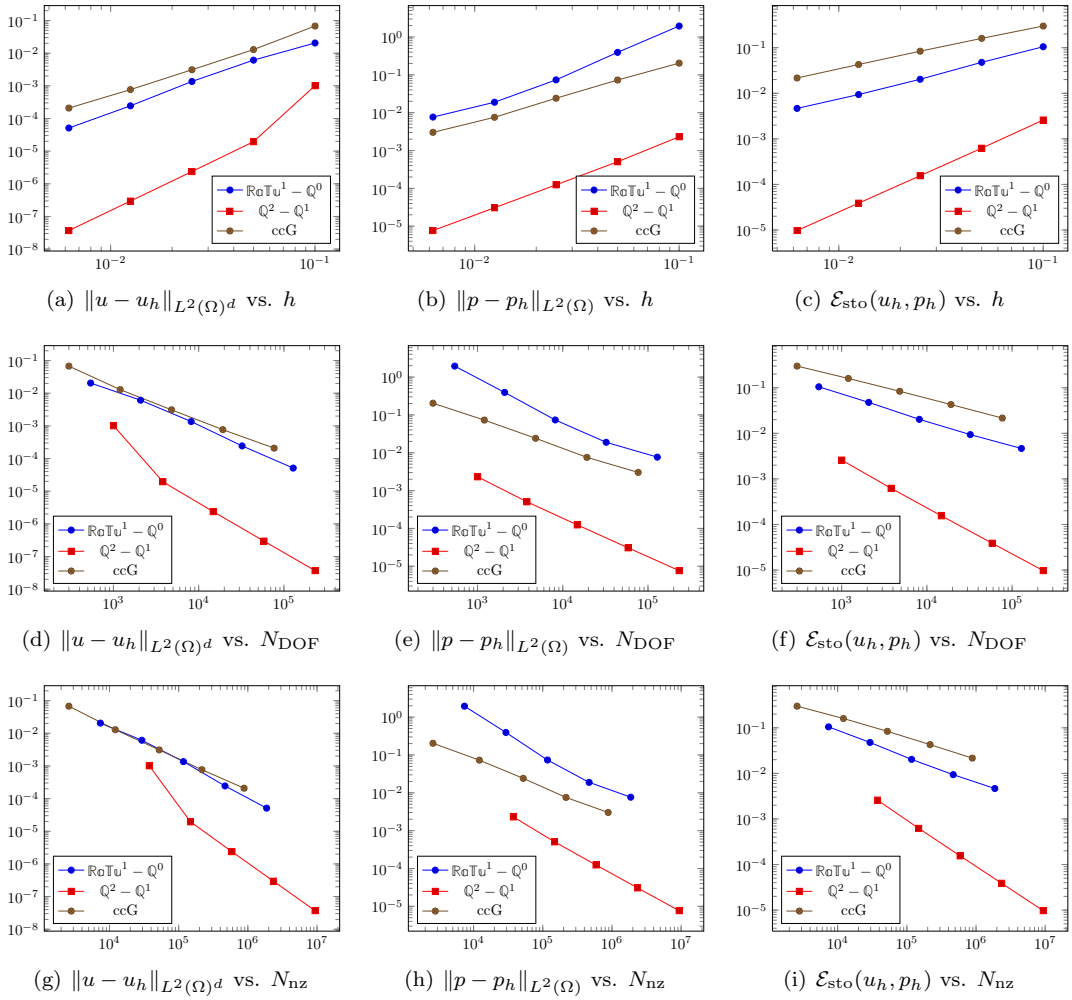
$$u_1(\mathbf{x}) = -\exp(x_1)(x_2\cos(x_2)+\sin(x_2)), \quad u_2(\mathbf{x}) = \exp(x)x_2\sin(x_2), \quad p(\mathbf{x}) = 2\exp(x_1)\sin(x_2)-\overline{p},$$

where $\overline{p}$ is chosen in such a way that the constraint (3.10d) is verified. The problem is solved on the skewed mesh family depicted in Figure 3.6(a). We compare the following methods: (i) the ccG method (3.11); (ii) an inf-sup stable method based on first-order Rannacher–Turek $\mathbb{R}\mathrm{t}\mathbb{T}\mathrm{u}^1$ elements for the velocity and $\mathbb{Q}^0$ elements for the pressure; (iii) an inf-sup stable method based on second-order $\mathbb{Q}^2$ elements for the velocity and $\mathbb{Q}^1$ elements for the pressure. The error is measured in terms of the L^2-norm for both the velocity and the pressure. The energy-norm of the error is defined as follows:

$$\mathcal{E}_{\mathrm{sto}}(u_h,p_h)^2 \stackrel{\mathrm{def}}{=} \|\nabla u - \nabla_h u_h\|_{L^2(\Omega)^{d,d}}^2 + \|p-p_h\|_{L^2(\Omega)}^2 + \alpha\left(\sum_{F\in\mathcal{F}_h} h_F^{-1}\|[\![u_h]\!]\|_{L^2(F)^d}^2 + \sum_{F\in\mathcal{F}_h^i} h_F\|[\![p_h]\!]\|_{L^2(F)}^2\right),$$

where $\alpha = 1$ for the ccG method and $\alpha = 0$ for both the $\mathbb{R}\mathrm{t}\mathbb{T}\mathrm{u}^1 - \mathbb{Q}^0$ and the $\mathbb{Q}^2 - \mathbb{Q}^1$ methods.

The accuracy and memory consumption analysis for the Stokes benchmark is provided in Figure 3.16. As expected, the higher-order method benefits from the regularity of the solution and is therefore more efficient. The results in terms of the L^2-error on the velocity are comparable for both the ccG and the $\mathbb{R}\mathrm{t}\mathbb{T}\mathrm{u}^1 - \mathbb{Q}^0$ methods, whereas the ccG method has a slight edge when it comes to the L^2-norm of the pressure. As regards the energy norm, the differences between the $\mathbb{R}\mathrm{t}\mathbb{T}\mathrm{u}^1 - \mathbb{Q}^0$ and the ccG methods are essentially related to the presence of the jumps of the pressure in the energy norm for the latter. An interesting remark is that the superconvergence phenomenon observed in the example of §3.5.4 for the ccG method is no longer present here.



(a) $\|u - u_h\|_{L^2(\Omega)^d}$ vs. h (b) $\|p - p_h\|_{L^2(\Omega)}$ vs. h (c) $\mathcal{E}_{\text{sto}}(u_h, p_h)$ vs. h

(d) $\|u - u_h\|_{L^2(\Omega)^d}$ vs. N_{DOF} (e) $\|p - p_h\|_{L^2(\Omega)}$ vs. N_{DOF} (f) $\mathcal{E}_{\text{sto}}(u_h, p_h)$ vs. N_{DOF}

(g) $\|u - u_h\|_{L^2(\Omega)^d}$ vs. N_{nz} (h) $\|p - p_h\|_{L^2(\Omega)}$ vs. N_{nz} (i) $\mathcal{E}_{\text{sto}}(u_h, p_h)$ vs. N_{nz}

Figure 3.16: Accuracy and memory consumption analysis for the example of Sect. 3.5.5

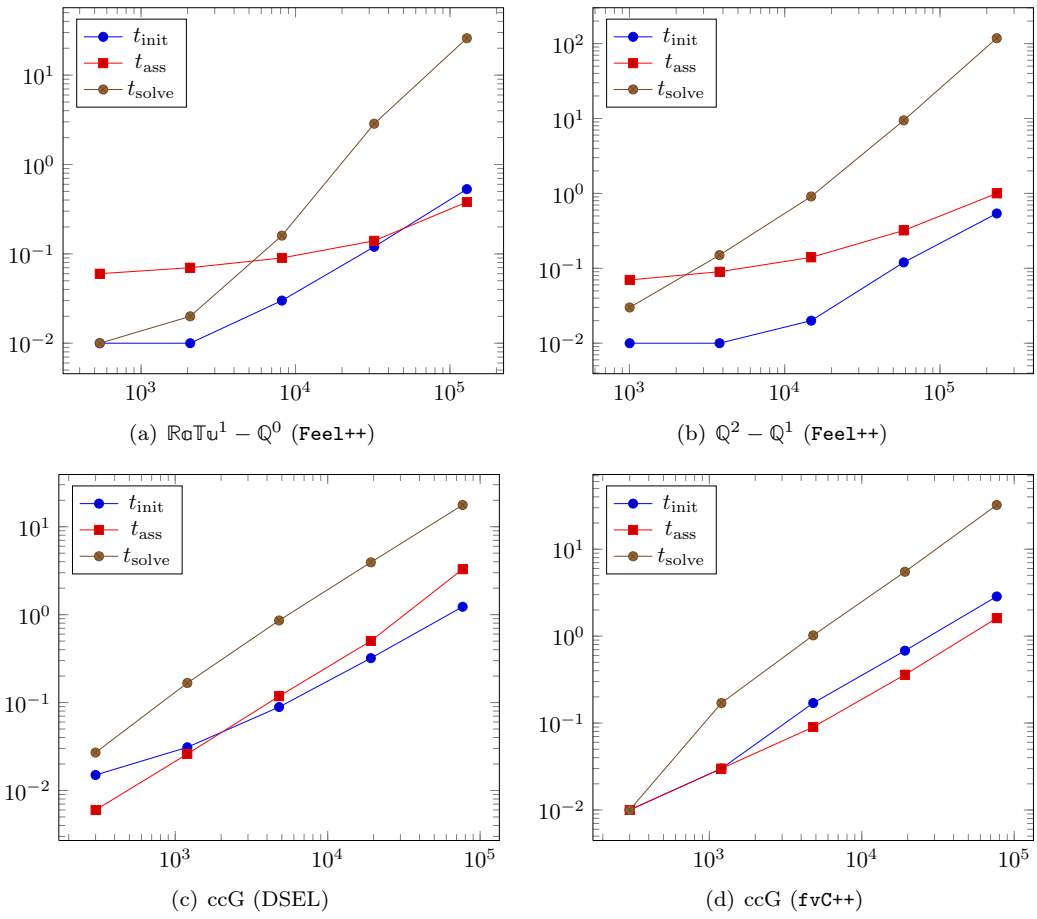
The performance analysis for the Stokes benchmark is provided in Figure 3.16. Similar considerations hold as for the benchmark of §3.5.4. In this case, however, t_{solve} largely dominates $t_{\text{init}} + t_{\text{ass}}$ (especially for the $\mathbb{Q}^2 - \mathbb{Q}^1$ method). The reason is that the chosen default linear solver is not necessarily the more appropriate. It has been excluded from the overall time comparison in Figure 3.18 to improve readability.

3.5.6 Multiscale methods

In this section, we present some results to validate the implementation of the multiscale method described in §2.5. We first study a simple 1D study case on which it is easy to analyze the numerical results and the effect of the coarse fine size grid ratio, and the effect of the homogenization of the permeability tensor. We then present a 2D study case on a unit square domain with an anisotropic permeability field. We analyze the convergence behaviour of the method and compare its performance to the SUSHI method.

1D test case

We solve the diffusion problem (2.17) on a 1D domain $\Omega = [0, 1]$ discretized by a first mesh with 100 cells and a second with 2048 cells. We consider the following boundary conditions:



(a) $\mathbb{RaTu}^1 - \mathbb{Q}^0$ (`Feel++`)

(b) $\mathbb{Q}^2 - \mathbb{Q}^1$ (`Feel++`)

(c) ccG (DSEL)

(d) ccG (`fvC++`)

Figure 3.17: Performance analysis for the example of §3.5.5

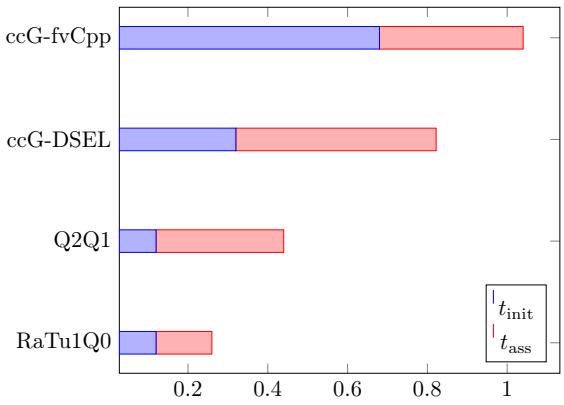


Figure 3.18: Comparison of different methods and implementations for the test case of §3.5.5

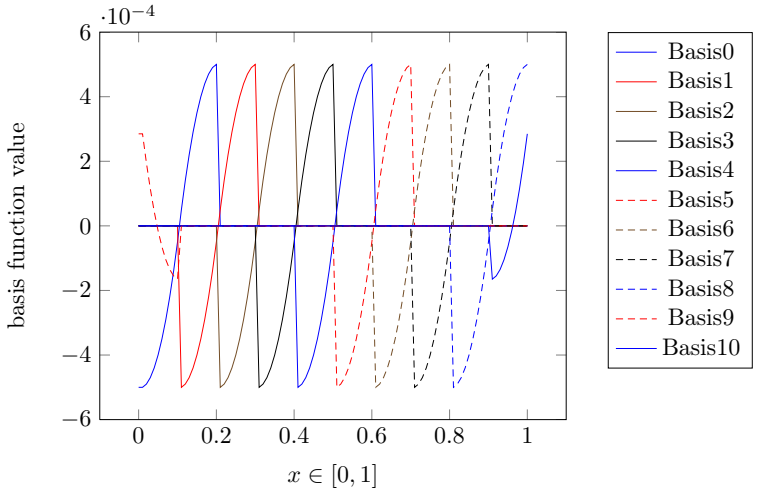


Figure 3.19: Plot of basis functions for $\kappa = 1$

- $u = -10$ on $\partial\Omega_{xmin}$;
- $u = 10$ on $\partial\Omega_{xmax}$;
- $\partial_n u = 0$ on $\partial\Omega \setminus \{\Omega_{xmin} \cup \Omega_{xmax}\}$

Let $size^f$ and $size^c$ be the number of cells of respectively the fine mesh and the coarse mesh, and $R = \frac{size^f}{size^c}$. We consider an homogeneous permeability field $\kappa_1 = 1$, a heterogenous field κ_2 with values between 0.1 (figure 3.20) and a log-normal field κ_3 with a spheric variogram with a correlated length $l_c = 0.3$ (figure 3.22). For each test we compute the reference solution with the SUSHI method on the fine grid. In Figure 3.19, we plot the value of the basis functions obtained with κ_1 and $R = 10$. In Figure 3.20 we compare for κ_1, the multiscale solution to the reference one. In Figure 3.21, we plot the value of the basis functions obtained with $size^f = 2018$, κ_3 and $R = 16$. In figure 3.22 we compare the multiscale solution to the reference one.

In Figures 3.19 and 3.21 we can easily check the support of each basis function. They effectively set a unit flux across their related coarse face. Figures 3.20 and 3.22 illustrate the behaviour of the method with respect to the heterogeneity of the permeability tensor κ. For the homogeneous test case, all the basis function have the same shape while for the heterogeneous test case, their shape are related to the values of the permeability tensor on the basis function support.

2D test case

We solve the diffusion problem (2.17) on a 2D unit square domain $\Omega = [-0.5, 0.5] \times [-0.5, 0.5]$. We consider the following boundary conditions:

- $u = -1$ on $\partial\Omega_{xmin}$;
- $u = 1$ on $\partial\Omega_{xmax}$;
- $\partial_n u = 0$ on $\partial\Omega \setminus \{\Omega_{xmin} \cup \Omega_{xmax}\}$

Let $size^f$ and $size^c$ be the number of cells of respectively the fine mesh and the coarse mesh, and the coarsening ratio $R = \frac{size^f}{size^c}$. We consider the family of coarse and fine meshes with $size^f \in \{20 \times 20, 40 \times 40, 80 \times 80\}$ and $R \in \{5, 10\}$. We consider the heterogenous field $\kappa(x) = \nu e^{\lambda \sin(\pi \frac{x}{\epsilon}) \sin(\pi \frac{y}{\epsilon}))}$ illustrated in Figure 3.23 for $\lambda = 1$ and $\epsilon = 0.35$. To have a reference solution, we solve the problem with the SUSHI method on a mesh of $size = 160 \times 160$ (cf. figure 3.24). For

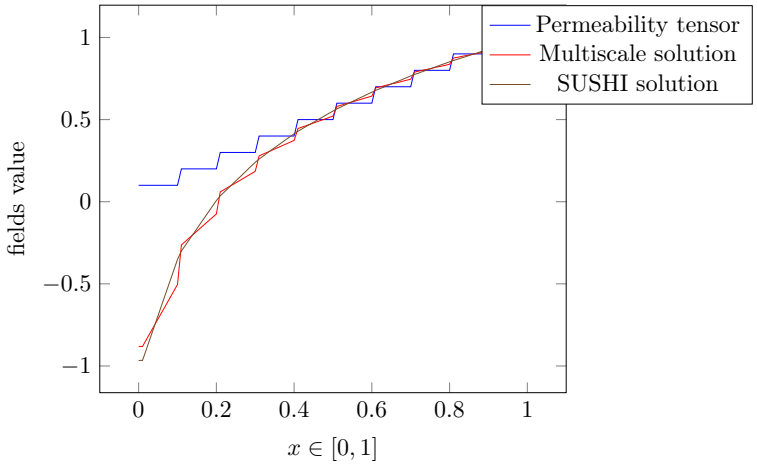


Figure 3.20: Heterogeneous permeability and Multiscale vs SUSHI solution

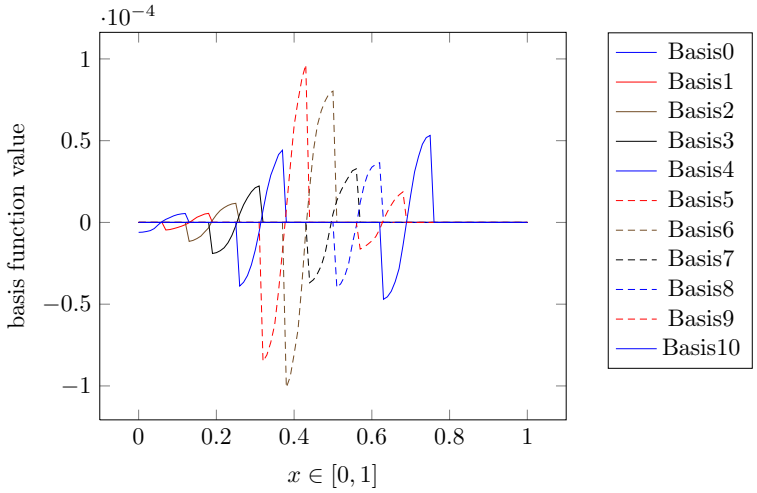


Figure 3.21: Basis functions

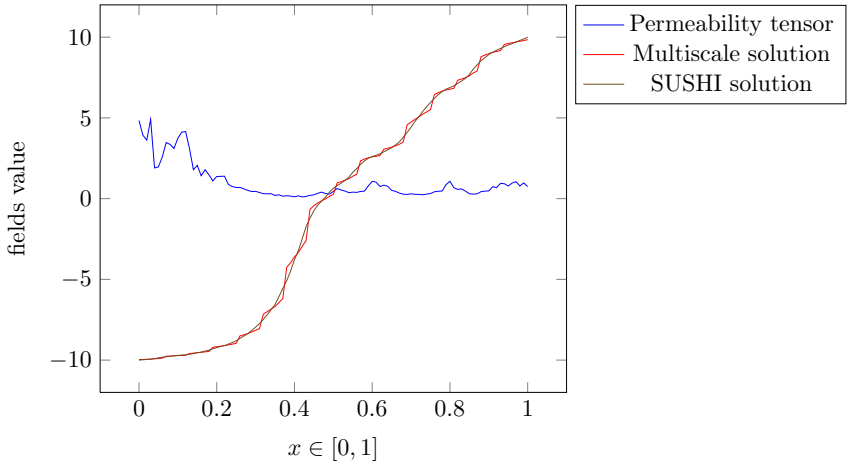


Figure 3.22: Heterogeneous permeability and Multiscale vs SUSHI solution

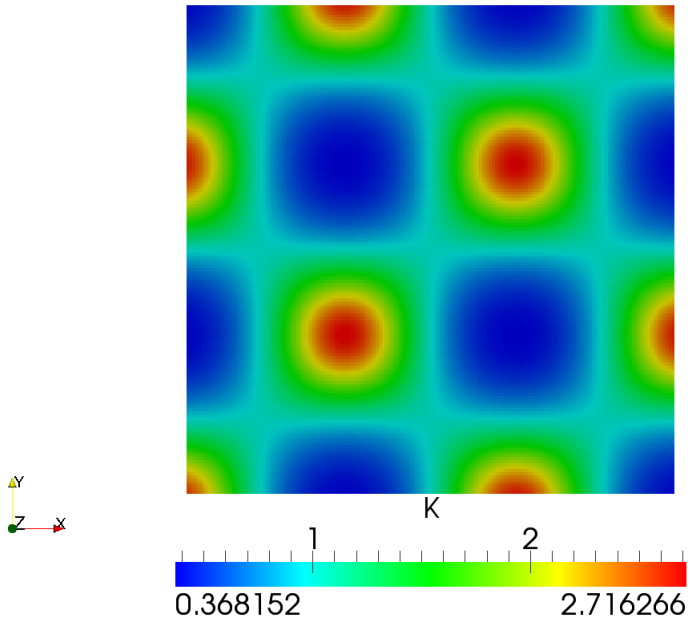


Figure 3.23: Heterogeneous permeability tensor field

each value of the parameters $size^f$ the fine grid size and R the coarsening ratio, we estimate the error of the solution by evaluating:

$$\frac{\|v_x - v_x^{ref}\|}{\|v_x^{ref}\|} + \frac{\|v_y - v_y^{ref}\|}{\|v_y^{ref}\|}$$

Tables 3.27 and 3.28 list the errors $\|v - v^{ref}\|$, $\|v_x - v_x^{ref}\|$, $\|v_x - v_x^{ref}\|$, $card(\mathcal{T}_h^f)$ the number of cells of the fine mesh, fine element size h^f and coarse element size h^c, respectively for $R = 5$ and $R = 10$. Tables 3.30, 3.31 list $t_{assembly}$, t_{solver}, $t_{downscale}$ and t_{total} the times in seconds to assemble the linear system, to solve it and to downscale the coarse solution on the fine mesh, respectively for $R = 5$ and $R = 10$. Table 3.32 list the these times for the SUSHI method on the fine mesh. These results show that on this synthetic test case, the method seems to converge to the same solution than the SUSHI method. For a given size of the thin grid, the cost of the multiscale method is smaller than the cost of the standard SUSHI method.

Figure 3.34(a) illustrates the permeability field for $\lambda = 4$ and $\epsilon = 0.35$. Figures 3.33(a) and 3.33(b) illustrate respectively the multiscale solution and the reference solution. Figure 3.34(b) illustrates the relative error between the multiscale solution and the reference one. Comparing the repartition of large error to the repartion of large scale variation of the permeability field, we can observe the numerical resonance effects of the multiscale method in regions concentrating large scale permeability variations. This well known behaviour is described by Yalchin R. Efendiev and Thomas Y. Hou and Xiao-Hui Wu in [62]. Zhiming Chen and Thomas Y. Hou propose in [44, 90] an over-sampling method consisting in building basis functions with overlapping supports. Other techniques consist in constructing judicious boundary conditions for the basis function problems (2.18) using for example the solution of first resolution of (2.17).

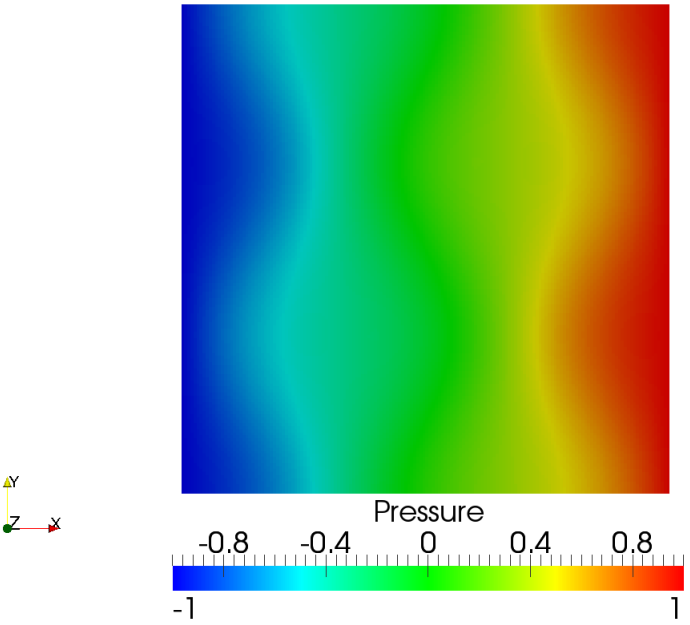


Figure 3.24: Reference pressure solution

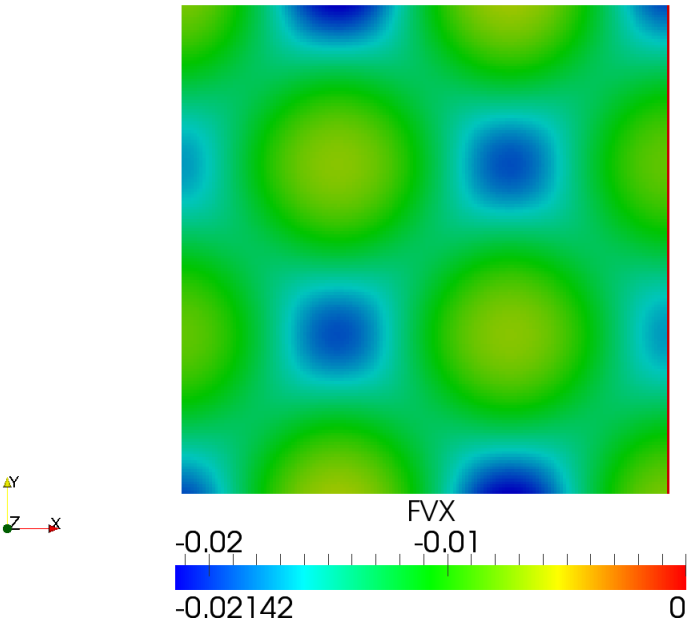


Figure 3.25: Reference velocity in X direction

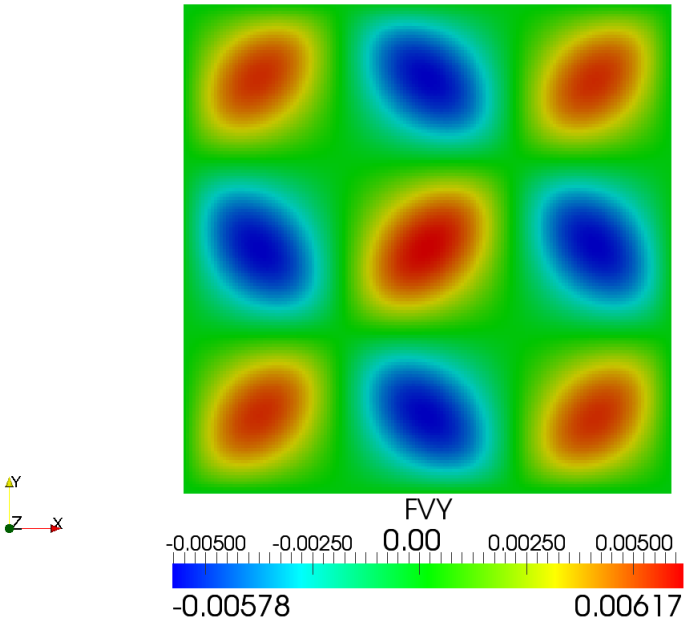


Figure 3.26: Reference velocity in Y direction

$card(\mathcal{T}_h^f)$	h^f	h^c	$\|v - v^{ref}\|$	$\|v_x - v_x^{ref}\|$	$\|v_x - v_x^{ref}\|$	order
20x20	0.05	0.25	1.81	1.04e-04	8.29e-06	
40x40	0.025	0.125	1.06	7.61e-05	1.16e-06	0.77
80x80	0.0125	0.0625	0.42	3.29e-06	6.76e-07	1.05

Figure 3.27: Convergence results for $R = 5$

$card(\mathcal{T}_h^f)$	h^f	h^c	$\|v - v_{ref}\|$	$\|v_x - v_x^{ref}\|$	$\|v_y - v_y^{ref}\|$	order
20x20	0.05	0.5	2.86	5.43e-04	8.29e-06	
40x40	0.025	0.25	1.82	1.02e-04	8.71e-06	0.65
80x80	0.0125	0.125	0.38	1.06e-06	7.60e-07	1.46

Figure 3.28: Convergence results for $R = 10$

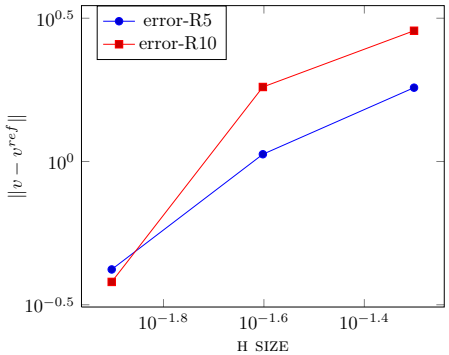


Figure 3.29: Convergence results: $\|v_v^{ref}\|$ vs h^f

$card(\mathcal{T}_h^f)$	h^f	h^c	$t_{assembly}$	t_{solver}	$t_{downscale}$	t_{total}
20x20	0.05	0.1	4.744e-04	4.158e-04	1.16e-03	8.91e-04
40x40	0.025	0.05	1.629e-03	1.602e-03	5.38e-03	3.23e-03
80x80	0.0125	0.025	1.046e-02	6.688e-03	2.26e-02	1.71e-02

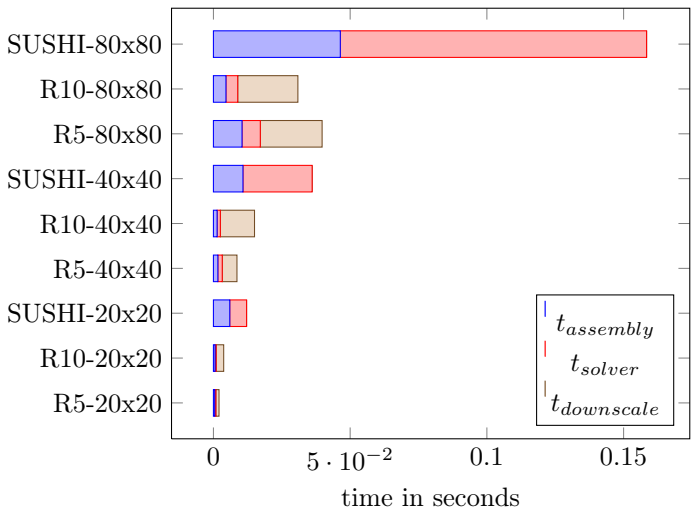
Figure 3.30: Convergence results for $R = 5$

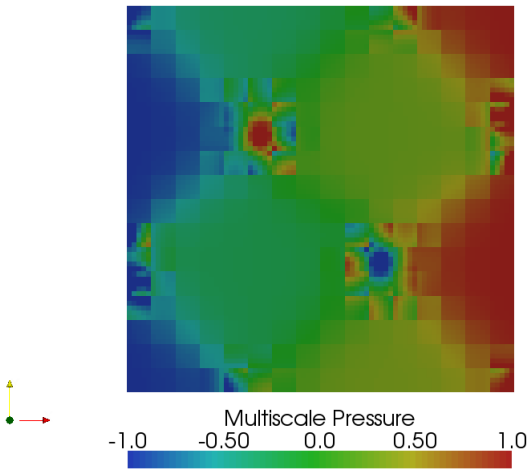
$card(\mathcal{T}_h^f)$	h^f	h^c	$t_{assembly}$	t_{solver}	$t_{downscale}$	t_{total}
20x20	0.05	0.5	5.95e-04	3.93e-04	2.77e-03	9.91e-04
40x40	0.025	0.25	1.35e-03	1.18e-03	1.25e-02	2.54e-03
80x80	0.0125	0.125	4.58e-03	4.31e-03	2.20e-02	8.90e-03

Figure 3.31: Convergence results for $R = 10$

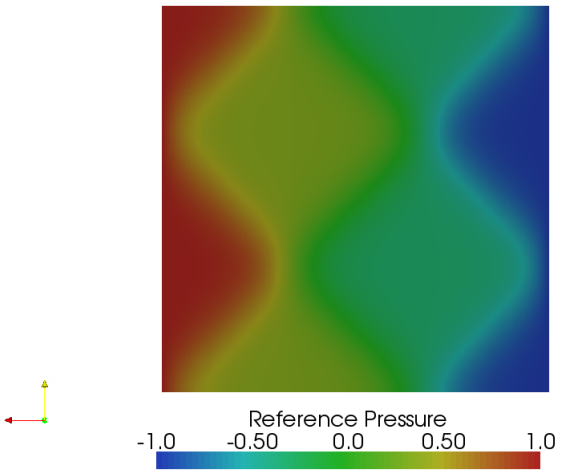
$card(\mathcal{T}_h^f)$	h^f	h^c	$t_{assembly}$	t_{solver}	$t_{downscale}$	t_{total}
20x20	0.05	0.5	5.98e-03	6.15e-03	0.	1.21e-02
40x40	0.025	0.25	1.08e-02	2.53e-02	0.	3.61e-02
80x80	0.0125	0.125	4.64e-02	1.12e-01	0.	1.59e-01

Figure 3.32: Performance results of the SUSHI scheme



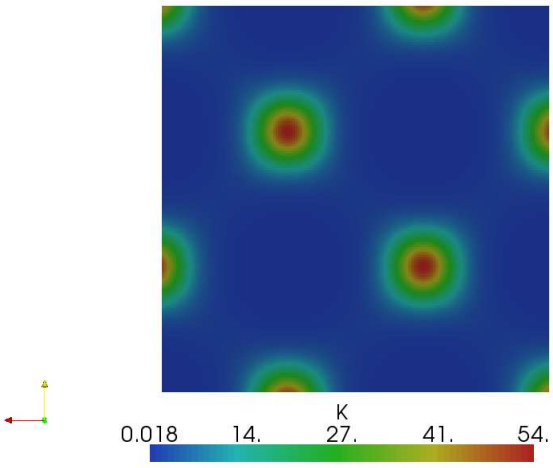


(a) Multiscale pressure

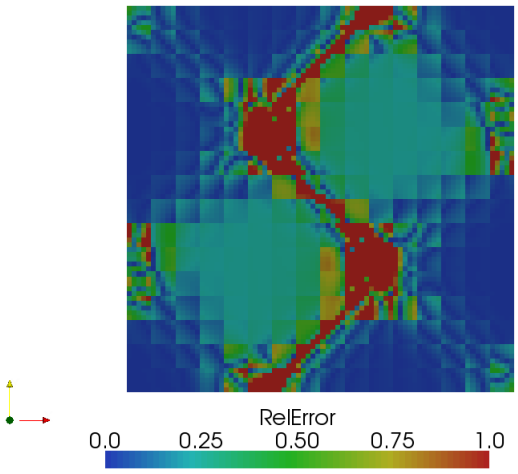


(b) Reference pressure

Figure 3.33: Mutiscale and the reference solution



(a) Domain description



(b) Relative error

Figure 3.34: Permeability field and relative error between the Mutiscale and the reference solution

Chapter 4

Runtime system for new hybrid architecture

4.1 Technical context and motivation

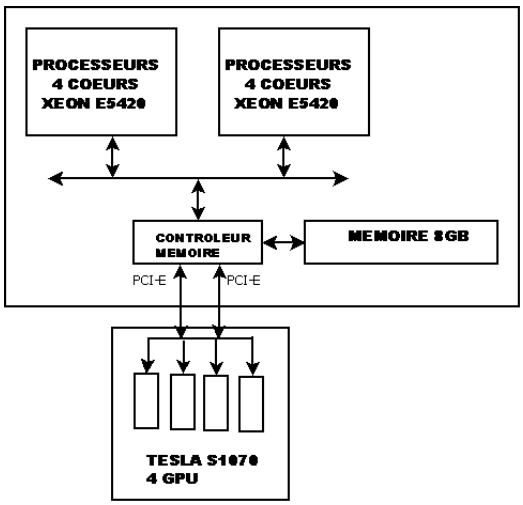
4.1.1 Hardware context

The trend in hardware technology is to provide hierarchical architecture with different levels of memory, process units and connexion between resources, using either accelerating boards or by the means of hybrid heterogeneous many-core processors. Nowadays the use of accelerators like Graphic cards (GPU), taking advantage of the gaming market and becoming more and more programmable, gives access to affordable process units providing a high level of parallelism for a limited power consumption. **Hybrid architectures** are based on clusters of hierarchical multi-core nodes where each node has a heterogeneous design using either accelerating boards or directly by the means of hybrid heterogeneous many-core processors. For example, the figure 4.1 illustrates the design of a SMP node with 2 quad-core processors connected to a GPU *tesla* server with 4 GPU. This kind of architecture, characterized by different levels of memory, heterogeneous computation units, gives access to a high potential amount of performance with different levels of parallelism: parallelism between the nodes of a cluster linked by a high speed network connexion, parallelism between the cores of shared memory nodes with multi-core processors linked by a SMP or a NUMA memory, or even at a higher level parallelism between all the cores of the streaming processors of accelerators like GP-GPU.

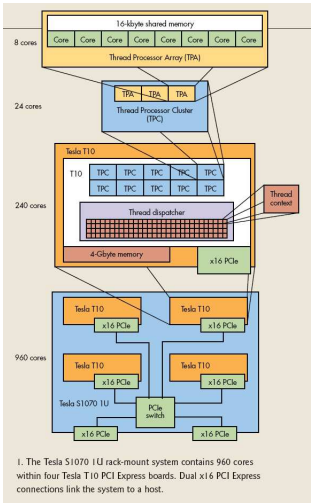
4.1.2 Programming environment for hybrid architecture

The complexity to handle hybrid architectures has considerably increased. The heterogeneity introduces serious challenges in term of memory coherency, data transfer between local memories, load balancing between computation units. All these issues are managed by the software layer, and the way they are handled becomes all the more important since the time scale of hardware evolution is much smaller than the time scale of software design. Various approaches have appeared to manage the different levels of parallelism involved to take advantage of the potential power of performance:

1. **Different programming models** have emerged to address parallelism:
    - the multi-threading paradigm is traditionally used for multi-core architecture with shared memory,
    - the message passing paradigm widely developed to support distributed memory architecture has been popularized by the emergence of the MPI standard,



(a) Herogeneous node



(b) GPU server

Figure 4.1: Example of hybrid architecture

- the data parallelism is in general used by solutions proposing a high level approach based on a distribution of the data (HPF),
- the task parallelism consisting in dividing the work into different pieces of computation (tasks) applied on a predefined data set is a paradigm that becomes more and more successfull with modern highly parallel architectures, as it provides a simple and flexible way to manage memory hierarchies and the parallelism of tasks, independent or organized into directed acyclic graphs.

2. **Programming environments** have been developed:
   - some are based on libraries provided to the end-user (Quark scheduler,TBLAS data management),
   - some provide compiling environments with generated compute kernel or annotation based language extension(TBB, OpenMP,..., HMPP, PGI, OpenACC,...),
   - OpenCL, OpenACC are emerging standards.
3. **Schedulers** have been developed to handle the problem of dispatching work on the different computation units available in an accelerator based platform. They are aimed at avoiding end users to map by their own tasks to devices. Different approaches exist, based on different kinds of technologies (syntax analysis, compiler, annotation-based languages,...), on which rely environments providing static, dynamic or mixed static-dynamic scheduling services more or less portable.
4. **Data management** support has become a critical issue as with distributed and hierarchical memory, data movement is often more expensive than computation. Providing a coherent memory subsystem or optimizing data transfer in order to ensure that data is available on time has become an important concern to have performance. There are two main approaches: explicit managed memory and the virtual distributed shared memory, a classical approach to deal with distributed memory consisting in implementing a Distributed Shared Memory (DSM) which provides programmers with a unified shared address space.
5. **Runtime systems** (see figure 4.2) provide higher-level software layers with convenient abstractions which permit to design portable algorithms without having to deal with low-level concerns. Compared to most of the other approaches, they provide support for both data management and scheduling. Among them there are research Runtime systems like:

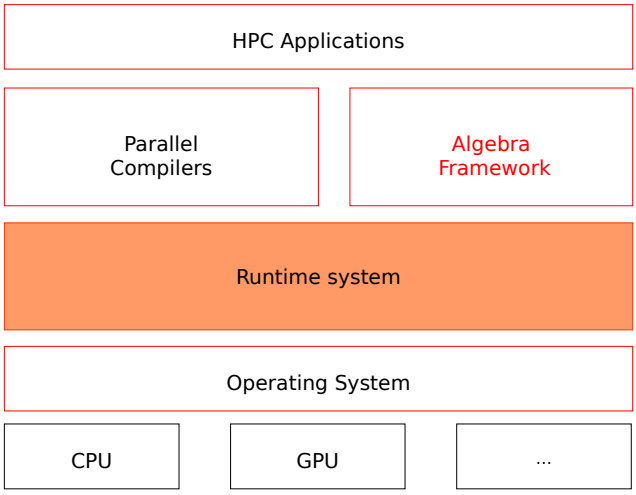


Figure 4.2: Runtime system for hybrid architecture

- Charm++ (Urbana, UIUC) a parallel C++ library that provides sophisticated load balancing and a large number of communication optimization mechanisms[1]
- StarSS, OmpSs (Barcelona, BSC) both a language extension and a collection of runtime systems targeting different types of platforms, that provide an annotation-based language which extends C or Fortran applications to offload pieces of computation on the architecture targeted by the underlying runtime system.
- StarPU (INRIA Bordeaux), that handles fine grain parallelism, provides dynamic scheduling of sequential tasks (DAG), performance prediction, implicit data management[35, 36].
- XKaapi (INRIA Rhones-Alpes) is a C++ library that allows to execute multi-threaded computation with data flow synchronization between threads[18].
- S_GPU (Bull, CEA INAC, UJF/LIG) aimed at coarse grain parallelism that provides services to map parallel tasks, for resource virtualization and explicit data management.
- HPX, an open source implementation of the ParalleX model for classic Linux based Beowulf clusters or multi-socket highly parallel SMP nodes, providing a runtime system architecture extendable to new computer system architectures. ([78])

4.1.3 Motivation for an Abstract Object Oriented Runtime System model

Multiscale methods like those described in Annexe C are based on algorithms providing a great amount of independent computations. For instance, in the method described in §2.5:

- the basis function computations lead to solve for N coarse faces F_c, the linear systems

$$\mathbf{A}_{F_c}\mathbf{x}_{F_c} = \mathbf{b}_{F_c}$$

  where $\mathbf{A}_{F_c}$ is a matrix, $\mathbf{x}_{F_c}$ and $\mathbf{b}_{F_c}$ are vectors;

- the computation of the elements of the coarse linear system is equivalent, for a coarse cell K_c and for two coarse faces $F_c \in \partial K_c$ and $F'_c \in \partial K_c$, to the algebraic computation

$$(\mathbf{G}_{Kc,F'_c}.\mathbf{x}'_{F'_c})^t.\mathbf{G}_{Kc,F_c}.\mathbf{x}_{F_c}$$

where $\mathbf{G}_{Kc,F_c'}$ are matrices, $\mathbf{x}_{F_c}$ vectors;

- the interpolation of the fine solution from the coarse solution leads to compute the following contributions

$$\mathbf{y}_{F_c} = v_{F_c}\mathbf{V}_{F_c}\mathbf{x}_{F_c}$$

where $\mathbf{V}_{F_c}$ are matrices, $\mathbf{x}_{F_c}$ vectors and v_{F_c} scalars.

Theses computations can be parallelized on various architectures with multiple nodes, mutiple cores and multiple accelerator boards with different kinds of technologies (MPI, TBB, OpenMP, CUDA, OpenCL...).

Such methods are good candidates to perform on new hardware technology. However, using often complex numerical concepts, they are developed by programmers that cannot deal with hardware complexity. Most of the approaches presented in §4.1.2 remain often too poor to manage the different levels of parallelism involved to take advantage of the potential of computing power. Runtime system solutions that expose convenient and portable abstractions to high-level compiling environments and to highly optimized libraries are interesting. Indeed they enable end users to develop complex numerical algorithms offering good performance on new hardware technology by hiding the low level concerns of data management and task scheduling. Such a layer provides a unified view of all processing units, enables various parallelism models (distributed data driven, task driven) with an expressive interface that bridges the gap between hardware stack and software stack.

In the DSEL framework presented in §3.3, we are led to generate C++ algorithms with objects of the back-end of the DSEL (§3.1). In this context an Object Oriented Runtime System solution is an interesting solution. Indeed such a layer is naturally integrated to `C++`, our host language, separating as illustrated in figure 4.2 the abstractions of the algebraic and numerical frameworks from those modeling the operating system and the various computation units (CPU, GPU) composing hybrid architectures. However, to handle the variety of new hybrid architectures and to follow the fast evolution of hardware design, its architecture needs to be based on the right abstract concepts so that it could be easily extended with new implementations modeling new hardware components, limiting in that way the impacts on our generative framework and separating the evolution of our language from the one of hardware.

Another important issue is to deal with legacy codes. Many existing Runtime System solutions involve important modifications in existing application architecture making painful the migration of legacy code to take advantage of the power of new hybrid hardware. Solutions that enable us to enhance specific parts of existing applications without needing to restructure the whole application architecture are interesting, as they can be deployed seamlessly in algorithms that can potentially perform well on hybrid architecture but are too complex to be re-written from scratch.

4.2 An abstract object oriented Runtime System Model

4.2.1 Contribution

In Chapter 3 we have presented a generative framework based on a DSEL that enables us to describe numerical methods at a high level, and a generative mechanism to generate `C++` codes with back-end objects. We propose a runtime system layer on top of which we base the back-end of the DSEL to obtain generated source code that performs efficiently on new heterogeneous hardware architectures. Our approach is to provide an abstract object oriented runtime system model that enables us to handle, in a unified way, different levels of parallelism and different grain sizes. Like for most existing Runtime System frameworks, the proposed model is based on:

- an abstract architecture model that enables us to describe in a unified way most of nowadays and future heterogeneous architectures with static and runtime information on the memory, network and computational units;
- an unified parallel model programing based on tasks that enables us to implement parallel algorithms for different architectures;
- an abstract data management model to describe the processed data, its placement in the different memory and the different way to access to it from the different computation units.

The main contribution with respect to existing frameworks is to propose an abstract architecture for the model based on **abstract concepts** . We define **Concept** as a set of requirements for types of objects that implement specific behaviours. Most of the abstractions of our Runtime system models are defined as requirements for `C++` structures. Algorithms are then written with some abstract types which must conform to the concepts they implement. This approach has several advantages:

1. it enables to clearly separate the implementation of the numerical layer from the implementation of the runtime system layer;
2. it enables to take into account the evolution of hardware architecture with new extensions, new concepts implementation, limiting in that way the impact on the numerical layer based on the DSEL generative layer;
3. it enables the benchmark of competiting implementations of each concept with various technologies, which can be based on existing research frameworks like StarPU which already provides advanced implementation of our concepts;
4. it enables us to design a non intrusive library, which unlike most of existing framework, does not constraint the architecture of the final applications. One can thus enhance any part of any existing applications with our framework, re-using existing classes or functions without needing to migrate the whole application architecture to our formalism. This issue is very important because often legacy codes cannot take advantage of new hybrid hardware because most of existing programming environments make the migration of such applications painful;
5. finally the proposed solution does not need any specific compiler tools and does not have any impact on the project compiling tool chain.

In this section we present the different abstractions on which the proposed framework relies. We detail the concepts we have proposed to modelize these abstractions. We illustrate them by proposing different types of implementation with various technologies. The back-end of our DSEL relies on this model, that bridges the gap between our DSEL and the low level API used to execute algorithms on various computational units. We study how the proposed solution enables us to address seamlessly heterogeneous architectures and to manage the available computation resources to optimize the application performance.

4.2.2 An abstract hybrid architecture model

The purpose of this abstract hybrid architecture model is to provide a unified way to describe hybrid hardware architecture and to specify the important features that enable to choose at compile time or at run time the best strategies to ensure performance. Such an architecture model has been already developed in the project **HWLOC** (Portable Hardware Locality)[10] which provides

> "a portable abstraction (across OS, versions, architectures, ...) of the hierarchical topology of modern architectures, including NUMA memory nodes, sockets, shared caches, cores and simultaneous multi-threading. It also gathers various system attributes such as cache and memory information as well as the locality of I/O devices such as network interfaces, InfiniBand HCAs or GPUs".

We propose an architecture model, based on the HWLOC framework. An architecture description is divided into two part, a static part grouping static information which can be used at compile-time and a dynamic part with dynamic information used at run-time. The information is modelized with the following abstractions:

- `system` which represents the whole hardware system;
- `machine` for a set of processors and memory with cache coherency;
- `node` modeling a NUMA node, a set of processors around the memory on which the processors can directly access.
- `socket` for sockets, physical packages, or chip;
- `cache` for cache memory (L1i, L1d, L2, L3,...);
- `core` for computation units;
- `pu` for processing unit, execution units;
- `bridge` for bridges that connect the host or an I/O bus, to another I/O bus;
- `pci_device` for PCI devices;
- `os_device` for operating system device.

The **static information** is represented by tag structures and string keys as in listing 4.1. They are organized in a tree structure where each node has a tag representing a hardware component, a reference to a parent node and a list of children nodes. Tag structures are used in the generative framework at compile time. For a target hardware architecture and with its static description, it is possible to generate the appropriate algorithms with the right optimisations.

The **dynamic information** is stored, in each node of the tree description, with a property map associating keys representing dynamic hardware attributes, to values which are evaluated at runtime, possibly using the `HWLOC` library. Theses values form the runtime information which enables to instantiate algorithms with dynamic optimization parameters like cache memory sizes, `stack_size` the size of the memory stack, `nb_pu` the maximum number of Process Units, `warp_size` the size of a NVidia WARP (NVidia group of synchronized threads) and `max_thread_block_size` the maximum size of a NVidia thread block executed on GP-GPUs, `nb_core` or `nb_gpu` the number of available physical cores of CPUs or GPUs,... Such runtime features, are useful parameters to optimize low level algorithms, in particularly for CUDA or OpenCL algorithms for GP-GPUs.

Such static and dynamic information associated to technics of tags dispatching is used by Joel Falcou in NT2 [12] to transform expressions at compile time and generate efficient code with right optimizations.

Listing 4.1: Tag id for hardware component units

```
namespace RunTimeSystem {
  namespace tag {
   struct system {
      static std::string name() { return std::string("system") ; }
    } ;
   struct machine {
      static std::string name() { return std::string("machine") ; }
    } ;
    struct core {
```

```
      static std::string name() { return std::string("core") ; }
    } ;
    struct pu {
      static std::string name() { return std::string("pu") ; }
    } ;
    struct cache {
      static std::string name() { return std::string("cache") ; }
    } ;
    /*  ... */
  }
}
```

4.2.3 An abstract unified parallel programming model

We propose an abstract unified parallel programming model based on the main following abstractions:

- a **task abstraction** representing pieces of work, or an algorithm that can be executed on a core or onto accelerators asynchronously. A task can have various implementations that can be executed more or less efficiently on various computational units. Each implementation can be written in various low level languages (C++, CUDA, OpenCL) with various libraries (BLAS, CUBLAS) and various compilation optimizations (SSE directives,...). Tasks can be independent or organized in direct acyclic graphs which represent algorithms.
- a **data abstraction** representing the data processed by tasks. Data can be shared between tasks and have multiple representations in each local memory device.
- a **scheduler abstraction** representing objects that walk along task graphs and dispatch the tasks between available computational units.

These abstractions are modelized with `C++` concepts (defined in §4.2.1). This approach enables to write abstract algorithms with abstract objects with specific behaviours. Behaviours can be implemented with various technologies more or less efficient with respect to the hardware on which the application is executed. The choice of the implementation can be done at compile time for a specific hardware architecture, or at runtime for general multi-platform application.

A particular attention has been paid in the design of the architecture to have a non intrusive solution in order to facilitate the migration of legacy code, to enable the reusability of existing classes or functions and to limit the impacts on the existing application architecture. The purpose is to be able to select specific parts of an existing code, for example some parts which a great amount of independent works, then to enhance them by introducing multi-core or gpu optimisation without having to modify the whole of the code.

Runtime System Architecture

The proposed runtime system architecture, illustrated in figure 4.3 is quite standard:

- Computation algorithms implemented by user free functions or classes are encapsulated in *Tasks* objects, managed by a centralized *task manager* ;
- The pieces of data processed by the task objects, represented by user data classes are encapsulated in *data handler* objects, managed by a centralized *data manager* ;
- The associations between tasks and the processed data handlers are managed by *DataArg* objects;
- Tasks are organized in DAGs and processed by *scheduler* objects that dispatch them on devices to run them with executing *drivers*.

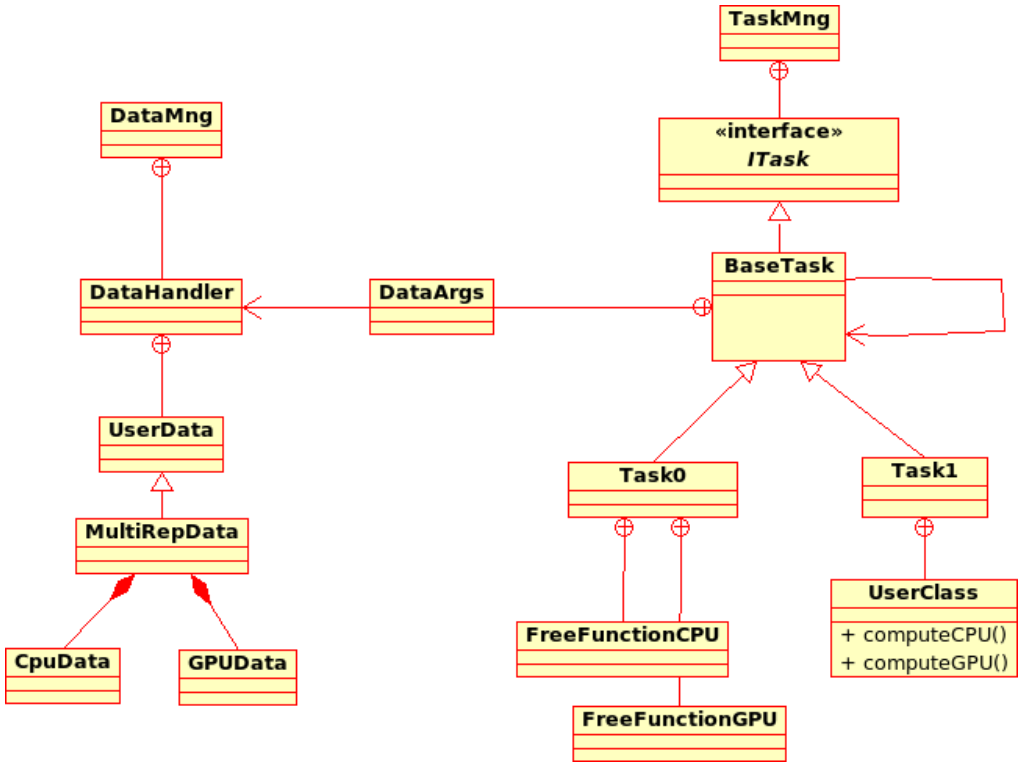


Figure 4.3: Runtime system architecture

The executing model is illustrated in figure 4.4:

- A *Scheduler* object processes a DAG of tasks belonging to a centralized task manger;
- *Task* objects which are ready to be executed are pushed back in a *task pool*;
- The *scheduler* object dispatches ready tasks on available computation devices, with respect to a given strategy;
- *Tasks* objects are executed on a target device by a *driver* object, then they are notified once their execution is finished;
- A *DAG* is completely processed once the task pool is empty.

Task management

The task management of our Runtime System Model is modelized with the class `TaskMng` described in listing 4.2. The sub type `TaskMng::ITask` is an interface class specifying the requirements for task implementation. `TaskMng::ITask` pointers are registered in a TaskMng object that associates them to an unique integer identifier `uid`. Tasks are managed in a centralized collection of tasks and dependencies between tasks are created with their `uid`. The base class `TaskMng::BaseTask` in listing 4.4 refines the `TaskMng::ITask` interface to manage a collection of uids of children tasks depending of the current task. Thus a **Directed Acyclic Graph** (DAG) (figure 4.5) is represented by a root task, and walking along it then consists in iterating recursively on each task and on its children.

Listing 4.2: TaskMng class

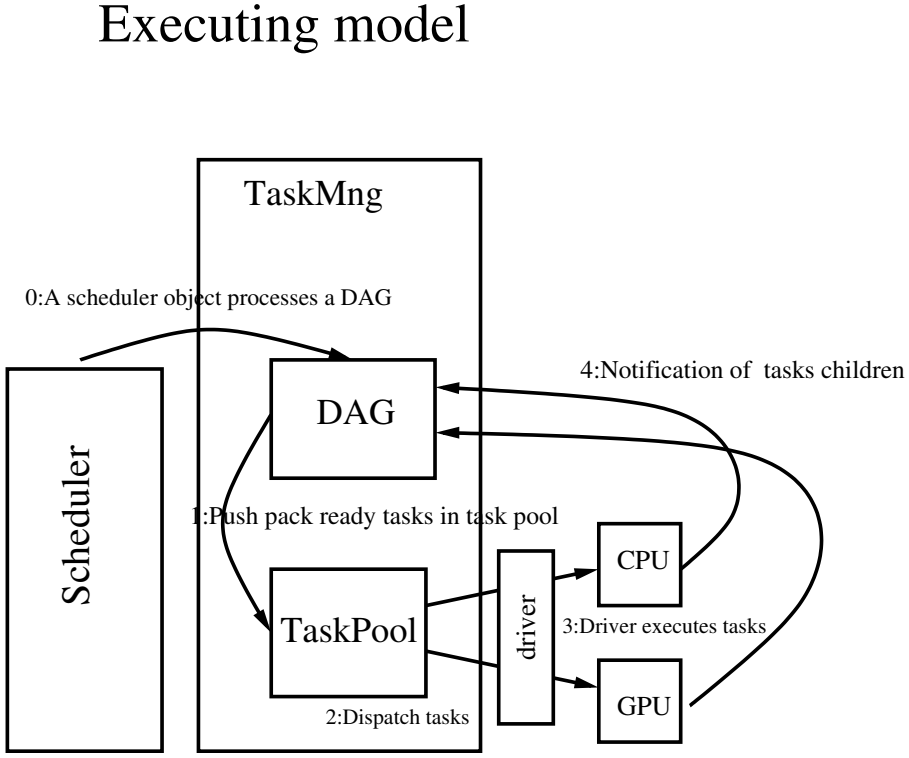


Figure 4.4: Executing model

```
class TaskMng {
public :
  typedef int uid_type ;
  static const int undefined_uid = -1 ;
  class ITask ;
  TaskMng(){}
  virtual ~TaskMng(){}
  int addNew(ITask* task) ;
  void clear() ;
  template<typename SchedulerT>
  void run(SchedulerT& scheduler, std::vector< int > const& task_list) ;
};
```

Listing 4.3: Task class interface

```
class TaskMng::ITask
{
public :
  ITask() : m_uid(TaskMng::undefined_uid){}
  virtual ~ITask() {}
  uid_type getUid() const {
    return m_uid ;
  }
  virtual void compute(TargetType& type,TaskPoolType& queue) = 0 ;
  virtual void compute(TargetType& type) = 0 ;
protected :
  uid_type              m_uid ;
```

```
} ;
```

Listing 4.4: Task class interface

```
class TaskMng::BaseTask : public TaskMng::ITask
{
public :
  BaseTask() ;
  virtual ~BaseTask() ;
  void addChild(ITask* child) ;
  void clearChildren() ;
  void notifyChildren(TaskPoolType& queue) ;
  void notify() ;
  bool isReady() const ;
};
```

The *Task* concept enables to implement a piece of algorithm for different kinds of target devices. A specific type of target device, or computational unit is identified by a unique *Target* label. Task instances are managed by a *TaskMng* that associates them to an unique id that can be used to create dependencies between tasks. Each task manages a list of children tasks. *Directed Acyclic Graphs* (DAGs) can be created with task dependencies. They have one root task. Task dependencies are managed by task unique id. To ensure graphs to be acyclic, tasks can only be dependent on an existing task with a lower unique id. A task can have various implementations. They are associated to a *Target* attribute representing the type of computational unit on which they should be used.

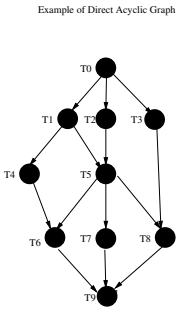


Figure 4.5: Example of directed acyclic graph

Data management

Our runtime system model is based on a centralized data management layer aimed to deal with:

- the data migration between heterogeneous memory units;
- an efficient data coherency management to optimize data transfer between remote memory and local memory;
- the concurrency of tasks accessing to shared data.

Our data management is based on the `DataMng` and `DataHandler` classes (listing 4.10). `DataHandler` objects represent pieces of data processed by tasks. They are managed by a `DataMng` object which has a `create` member function to instanciate them. `DataHandler` objects have a unique `DataUidType` identifier `uid`. The `DataArgs` class is a collection of `std::pair<DataHandler::uid_type,eAccessMode>` where `AccessMode` is an `enum` type with the following values `W`, `R` or `RW`. The `DataHandler` class provides a `lock`, `unlock` service to prevent data access concurrency:

- a task can be executed only if all its associated data handlers are unlocked ;
- when a task is executed, the DataHandlers associated with a `W` or `RW` mode are locked during execution and unlocked after.

A piece of data can have multiple representations on each device local memory. The coherency of all representations is managed with a timestamp DataHandler service. When a piece of data is modified, the timestamp is incremented. A representation is valid only if its timestamp is up to date. When a task is executed on a specific target device, the local data representation is updated only if needed, thus avoiding unuseful data transfer between different local memories.

Task dependencies

Task dependencies can be created in three ways:

- **Explicit task dependencies** is based on task uids. The `addChild` member function enables to create dependencies between tasks. Only a task with a lower uid can be the parent of another one, thus ensuring that the created graph is acyclic;
- **Logical tag dependencies**, based on task tags create dependencies between a group of tasks with a specific tag and another group of tasks with another specific tag;
- **Implicit data driven dependencies** is based on the sequential consistency of the DAG building order. When a task is registered, if the DataHandler access is in:
  - `RW` or `W` mode, then the task implicitly depends on all tasks with a lower uid accessing that same DataHandler in `R` or `RW` mode,
  - `R` or `RW` mode, then the task implicitly depends on the last task accessing that data in `RW` or `W` mode.

Once a task is executed, all its children tasks are notified. Each task manages a counter representing the number of parent tasks. When a task is notified, this counter is decremented . A task is ready when its parent counter is equal to zero and when all its dependent data handlers are unlocked. Its uid is then put in the queue of ready tasks managed by the scheduler that processes the DAG.

Scheduling and executing model

On heterogeneous architectures, the parallelism is based on the distribution of tasks on available computation units. The performance of the global execution depends a lot on the strategy used to launch independent tasks. It is well known that there is not a unique nor a best scheduling policy. The performance depends on both the algorithm and the hardware architecture. To implement various scheduling solutions adapted to different algorithms and types of architecture, we propose a `Scheduler` concept defining the set of requirements for scheduler types to represent scheduling models. The purpose of objects of such a type is to walk along task DAGs, to select and execute independent tasks on the available computation units, with respect to a given strategy. The principles for a scheduler object are:

1. to manage a pool of ready tasks (tasks which all parent tasks are finished and all datahandlers of its `DataArgs` attribut are unlocked);
2. to distribute the ready tasks on the different available computation units following a given scheduling stategy;
3. to notify the children tasks of a task once the task execution is finished;
4. to push back tasks that get ready in the pool of ready tasks.

The **TaskPoolConcept** defines the behaviour that must implement a type representing a `TaskPool`, that is to say the possibility to push back new ready tasks and to grab tasks to execute.

Listing 4.5: TaskPool

```
class TaskPoolConcept
{
public:
  template<typename TaskT>
  void pushBack(TaskT::uid_type uid) ;

  template<typename TaskT,typename TargetT>
  typename Task::ptr_type grabNewTask(TargetT const& target) ;

  bool isEmpty() const ;
};
```

A coarse grain parallelism strategy consists in executing the different independent ready tasks in parallel on the available computation units. We have implemented various schedulers like the `StdScheduler`, the `TBBScheduler` and `PoolThreadScheduler` described in §4.2.3

Parallelism can be managed at a finer grain size with concepts like the **ForkJoin** and the **Pipeline** concepts.

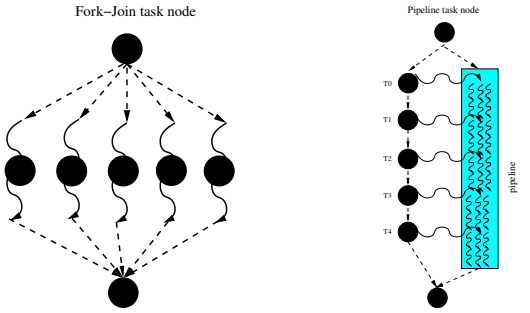
ForkJoin On multi-core architecture, a collection of equivalent tasks can be executed efficiently with technolgies like TBB, OpenMP, Posix threads. The ForkJoin concept (figure 4.6) consists in creating a DAG macro task node which holds a collection of tasks. When this node is ready, the collection of nodes is processed by a `ForkJoin Driver` in parallel. The macro task node is finished when all its children tasks are finished. The `ForkJoin Driver` is a concept defining the requirement for the types of objects that implement the fork-join behaviour with different technologies or libraries like TBB, Boost.Thread or Pthread.

Listing 4.6: Fork-Join driver concept

```
class ForkJoinDriverConcept
{
public:
  template<typename TaskT,typename TargetT, typename QueueT>
  void execForkJoin(std::vector< typename TaskPtrT::ptr_type > const& tasks,
                    std::vector< typename TaskPtrT::uid_type > const& uids,
                    TargetT& target,
                    QueueT& queue) ;

  template<typename TaskT,typename TargetT>
  void execForkJoin(std::vector< typename TaskPtrT::ptr_type > const& tasks,
                    std::vector< typename TaskPtrT::uid_type > const& uids,
                    TargetT& target) ;
};
```

Pipeline On vectorial device or accelerator boards, the **Pipeline** concept (figure 4.6) consists in executing a sequence of tasks (each task depending on its previous one) with a specific internal structure of instructions. The `Pipeline Driver` is a concept defining the requirement for the types of objects implementing the pipeline behaviour. These objects are aware of the internal structure of the tasks and execute them on the computation device in a optimized way often with a thin grain size parallelism. This approach is interesting for new GPU hardwares which can execute concurrent kernels. It enables to implement optimized algorithms with streams and



(a) ForkJoin node (b) Pipeline node

Figure 4.6: ForkJoin and pipeline task node

asynchroneous execution flows that improve the occupancy of device resources and lead then to better performance. For instance, for the computation of the basis functions of the multiscale model, we illustrate in §4.4 how the flow of linear system resolutions can be executed efficiently on GPU device with the GPUAlgebraFramework layer.

Listing 4.7: pipeline driver concept

```
class ForkJoinDriverConcept
{
public:
  template<typename TaskT,typename TargetT, typename QueueT>
  void execPipeline(std::vector< typename TaskPtrT::ptr_type > const& tasks,
                    std::vector< typename TaskPtrT::uid_type > const& uids,
                    TargetT& target,
                    QueueT& queue) ;

  template<typename TaskT,typename TargetT>
  void execPipeline(std::vector< typename TaskPtrT::ptr_type > const& tasks,
                    std::vector< typename TaskPtrT::uid_type > const& uids,
                    TargetT& target) ;
};
```

Asynchronism management On an architecture with heterogeneous memories and computation units, it is important to provide enough work to all available computation units and to reduce the latency due to the cost of data transfer between memories. The Asynchronism mechanism is a key element for such issues. The parametrized classes
`template<typename DriverT,typename TaskT> class AsynchTask` and
`template<typename AsynchTaskT> class Wait` implement the asynchronous behaviour:

- the `AsynchTask<TaskT>` is a task node that executes asynchronously its child task;
- the `Wait<AsynchTaskT>` is a task node that waits for the end of the execution the child task of the previous node then notifies the children of this task.

The `Driver` concept specifies the requirement of the type of objects that implement the asynchronous behaviour. This behaviour can be easily implemented with threads. The child task is executed in a thread. The end of the execution corresponds to the end of the thread. For GPU device, this behaviour can be implemented using a stream on which is executed an asynchronous kernel. The `wait` function is implemented with a synchronisation on the device.

The asynchronous mechanism is interesting to implement data prefetching on device with remote memory. Prefetch task nodes can be inserted in the DAG to load asynchronously data on GPU device so that they are available when the computational task is ready to run.

Listing 4.8: Data management

```
template<typename DriverT,typename TaskT>
class AsynchTask : public TaskMng::BaseTask
{
public:
  typedef TaskMng::BaseTask   BaseType ;
  AsynchTask(DriverT& driver,TaskT& task) ;
  virtual ~AsynchTask() ;
  virtual void wait(TargetType& type,TaskPoolType& queue) ;
  virtual void notify() ;
  virtual bool isReady() const ;
  void compute(TargetType& type) ;
  void compute(TargetType& type,TaskPoolType& queue) ;
  void finalize(TargetType& type,TaskPoolType& queue) ;
private :
  TaskT& m_task ;
};

template<typename AsynchTaskT>
class Wait : public TaskMng::BaseTask
{
public :
  typedef TaskMng::BaseTask   BaseType ;
  Wait(AsynchTaskT& parent) ;
  virtual ~Wait() ;
  void compute(TargetType& type,TaskPoolType& queue) ;
  void compute(TargetType& type) ;
  void finalize(TargetType& type,TaskPoolType& queue) ;
private :
  AsynchTaskT& m_parent ;
} ;
```

Example of application of the runtime system

With our runtime system abstractions, listing 4.9 illustrates how to write a simple program adding two vectors, which can be executed on various devices.

Listing 4.9: Simple vector addition program

```
class AxpyTask {
   void computeCPU(Args const& args) {
     auto x const& args.get<VectorType>('x').impl<tag::cpu>();
     auto y& args.get<VectorType>('y').impl<tag::cpu>();
     SAXPY(x.size(), 1.0, x.dataPtr(), 1, y.dataPtr(), 1);
   }
   void computeGPU(Args const& args) {
     auto x const& args.get<vector_type>('x').impl<tag::gpu>();
     auto y& args.get<vector_type>('y').impl<tag::gpu>();
     cublasSaxpy(x.size(), 1.0, x.dataPtr(), 1, y.dataPtr(), 1);
     cudaThreadSynchronize();
   }
} ;

int main(int argc, char **argv) {
```

```
    float vec_x[N], vec_y[N];

    /*(...)*/
    //
    // DATA MANAGEMENT SET UP
    DataMng data_mng ;
    VectorType x ;
    VectorType y ;
    DataHandler* x_handler = data_mng.create<VectorType>(&x) ;
    DataHandler* y_handler = data_mng.create<VectorType>(&y) ;

    //
    // TASK MANAGEMENT SET UP
    TaskMng task_mng ;
    /*(...)*/
    AxpyTask op ;
    TaskMng::Task<AxpyTask>* task = new TaskMng::Task<AxpyTask>(op) ;
    task->set<tag::cpu>(&AxpyTask::computeCPU) ;
    task->set<tag::gpu>(&AxpyTask::computeGPU) ;
    task->args().add('x',x_handler,ArgType::mode::R) ;
    task->args().add('y',y_handler,ArgType::mode::RW) ;

    int uid = task_mng.addNew(task) ;
    task_list.push_back(uid) ;

    //
    // EXECUTION
    SchedulerType scheduler ;
    task_mng.run(scheduler,task_list) ;
}
```

Elements of implementation of different concepts

Data and task management concepts The implementation of data and task management is based on the following principles:

- User Data are implemented by the mean of user `C++` classes or structures;
- User algorithms are implemented by the means of user free functions or member functions of user `C++` classes.

We have implemented `DataHandler` as a class that encapsulate any user classes or structures and which provides functions to retrieve the original user data structure, to `lock` or `unlock` the user data.

The `DataMng` is a centralized class that manages a collection of `DataHandler` objects and their integer unique identifier. This class enables to access any user data by the means of its unique identifier.

Listing 4.10: Data management

```
typedef enum {R,W,RW,Undefined} eAccessModeType ;

class DataHandler
{
public:
```

```
    typedef int       uid_type ;
    static const int null_uid = -1 ;
    DataHandler(uid_type uid=null_uid) ;
    virtual ~DataHandler() ;
    uid_type getUid() const ;
    template<typename DataT>
    DataT* get() const ;
    void lock() ;
    void unlock() ;
    bool isLocked() const ;
  };

  class DataMng
  {
  public :
    typedef DataHandler* DataHandlerPtrType ;
    DataMng() ;
    virtual ~DataMng();
    template<typename DataT>
    DataHandler* create() ;
    DataHandler* getData(int uid) const ;
  };
```

Tasks are implemented with the classes `TaskMng::Task0` and `class TaskMng::Task` (listing 4.11). `TaskMng::Task0` encapsulates any user free function while `class TaskMng::Task`, parametrized by a type `ComputerT` encapsulates a user class `ComputerT` and its member function, stored in a `boost::function` attribute. They implement the `TaskMng::ITask` interface that enables any scheduler to execute task objects on any target computation unit. They have a `set(<target>,<function>)` member function to define the implementation of the task for each target device.

Listing 4.11: Task class inplementation

```
class TaskMng::Task0 : public TaskMng::BaseTask
{
public :
  typedef ITask::TargetType                    TargetType ;
  typedef boost::function1<void,
                           DataArgs const&>    FuncType ;
  typedef std::map<TargetType,FuncType>        FuncMapType ;
  typedef typename FuncMapType::iterator       FuncIterType ;
  Task0() ;
  virtual ~Task0() ;
  void set(TargetType type,FuncType func) ;
  void compute(TargetType& type,TaskPoolType& queue) ;
  void compute(TargetType& type) ;
} ;

template<typename ComputerT>
class TaskMng::Task : public TaskMng::BaseTask
{
public :
  typedef ITask::TargetType                    TargetType ;
  typedef boost::function2<void,
                           ComputerT*,
                           DataArgs const&>    FuncType ;
  typedef std::map<TargetType,FuncType>        FuncMapType ;
  typedef typename FuncMapType::iterator       FuncIterType ;
```

```
  Task(ComputerT* computer) ;
  virtual ~Task() ;
  void set(TargetType type,FuncType func) ;
  void compute(TargetType& type,TaskPoolType& queue) ;
  void compute(TargetType& type) ;
} ;
```

Task execution When a task is executed, data user structures are recovered with a `DataArgs` object that stores data handlers and their access mode. This data can be locked if it is accessed in a write mode when the user algorithm is applied to it. Modified data is unlocked at the end of the algorithm execution.

TaskPool concept We have implemented the `TaskPoolConcept` with a simple parametrized class `template<TaskMng> class TaskPool` with two attributes: `m_uids` a collection of task uid and `m_mng` a reference to the task manager. The member function `pushBack(TaskMng::ITask::uid_type uid)` feeds the collection of ready tasks. The `Task::ptr_type grabNewTask(<target>)` grabs a `uid` from `m_uids` and returns the corresponding task with `m_mng`.

Scheduler concept To implement a scheduler class, one has to implement the `exec(<tasks>,<list>)` function that gives access to a collection of tasks and a list of tasks, roots of different DAGs. Walking along these DAGs, the scheduler manages a pool of ready tasks: the scheduler grabs new tasks to execute, children tasks are notified at the end of execution and feed the task pool when they are ready. Some `Driver` objects can be used to execute tasks on specific devices, to modelize different parallel behaviours, to give access for example to a pool of threads that grab tasks to be executed in the pool of ready tasks. We have implemented the following scheduler types:

- the `StdScheduler` is a simple sequential scheduler executing the tasks of a TaskPool on a given target device;
- the `TBBScheduler` is a parallel scheduler for multi-core architecture implemented with the `parallel_do` functionality of the TBB library;
- the `PoolThreadScheduler` is a parallel scheduler based on a pool of threads dispatched on several cores of the multi-core nodes, implemented with the `Boost.Thread` library. Each thread is associated to a physical core with an affinity, and dedicated to executed tasks on this specific core or on an accelerator device. The scheduler dispatches the tasks of the task pool on the threads which are starving.

ForkJoinDriver implementation We have developed for multi-core architectures three implementations conforming to this concept:

- the `TBBDriver` is a multi-thread implementation using the `parallel_for` algorithm of the TBB library;
- the `BTHDriver` is a multi-thread implementation based on a pool of threads implemented with the `Boost.Thread` library;
- the `PTHDriver` is a multi-thread implementation based on a pool of threads written with the native posix thread library.

4.3 Parallelism and granularity considerations

Hybrid architectures provide different levels of parallelism, parallelism between nodes with distributed memory and linked by a high speed connexion network, parallelism inside nodes with more and more cores per processor with shared memory, and at least the parallelism inside accelerators with a great amount of physical computation units and even more available processing units like threads. It is important to optimize each level of parallelism, maximizing the computation units occupancy keeping all of them busy, feeding them with both data and work. In this section, we see how the proposed runtime system can be a solution to deal with the different levels of parallelism. We first detail the parallelization principles for multi nodes with distributed memory architecture, then we study different levels of parallelism inside a SMP or NUMA node, the granularity of the tasks and different ways to deal with tiny tasks avoiding the overhead of task creation and destruction.

4.3.1 Parallelisation on distributed architecture

The first level of parallelization of our framework concerns the parallelization of global PDE problems described and solved with our DSEL. This level of parallelization enables us to solve large problems on distributed memory architectures with a limited memory size per node by distributing data between nodes. For such a parallelism, we have a standard approach based on the message passing paradigm, well adapted for this kind of architecture, based on a partition of the mesh among processors. Data are distributed with respect to the mesh distribution and computation works, with respect to the data they proccess. In §3.3.4 we have introduced space constraints and extra closure equations to deal with boundary conditions. In this section we discuss the way we parallelize these extra conditions. Finally we also discuss the parallelization of the CCG method which needs a special treatment: the use of **jump** operator on faces in the discrete formulation of this method leads to manage large linear combinations that leading to difficulties on faces shared by different mesh domains.

Mesh, data and linear system distribution

Our DSEL is based on the Arcane framework that provides mesh partition services. When the mesh data $\mathcal{T}_h$ is loaded, it is partitioned in n_p sub-domains $P_{i\in\{0,\dots,n_p-1\}}$ with standard partitioner algorithms (Parmetis, Scotch, Zoltan,...). Subdomains P_i are distributed on MPI processes of rank i. For each subdomain P_i, we define the set of ghost cells connected by faces as the set of cells $\tau_j \in P_{j,j\neq i}$ such that $\exists \tau_i \in P_i / \tau_i \cap \tau_j \neq \emptyset$. These ghost cells are added to each MPI subdomains. The set of nodes and faces connected to those cells are also designated as ghost mesh entities. Let **MPI subdomain** be the set of mesh entities processed by a MPI process of rank i, i.e. the subdomain P_i and its ghost entities. For each MPI subdomain, we define the **physical boundary** as the set of faces of P_i belonging to $\mathcal{F}_h^b$, the **MPI interface** as the set of faces of the boundary of the **MPI subdomain** not belonging to its **physical boundary** and the **physical interface** between two subdomains P_i and P_j as the set of faces shared by P_i and P_j. In figure 4.7 we illustrate a mesh partitioned into two sub-domains P_1 and P_2 and the **physical boundary** of P_1 in black, the **physical interface** in blue and the **MPI interface** in red.

Discrete variables (presented in 3.1) are distributed with respect to the mesh distribution. That is to say their values are indexed only by the mesh entities managed by the current MPI process. The coherency of data shared by several processes, the values indexed by ghost entities, is ensured with synchronization services of the Arcane framework presented in §1.2.2. In our framework, DOFs are deeply linked to mesh entities as each of them refer to its parent mesh entity, and the distribution of vector of DOFs follows naturally the distribution of the values of the discrete variables on which they are based. Let **local DOFs** be the DOFs related to local mesh entities and **ghost DOFs**, DOFs related to ghost entities. The vectors of our backend (presented in §3.1) are algebraic representations of vectors of DOFs. Finally linear system distribution follows as well the

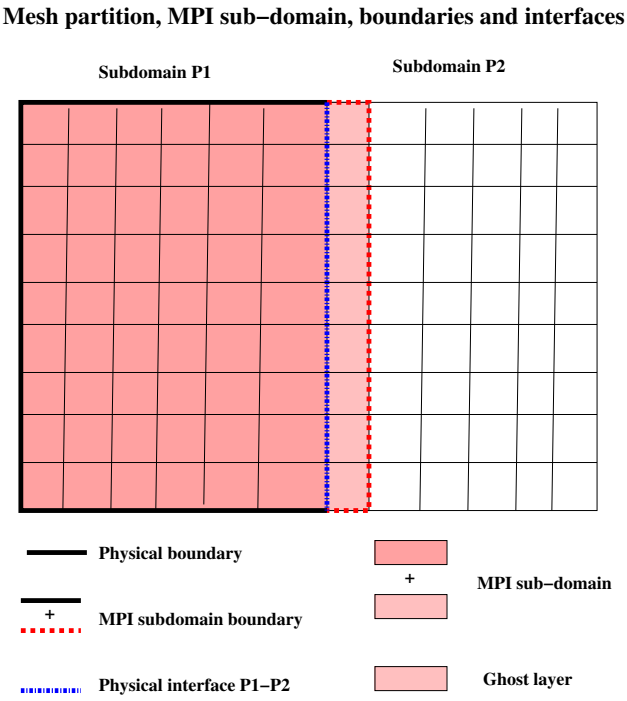


Figure 4.7: Mesh partition, sub-domain boundaries and interfaces

DOFs distribution (figure 4.8). As for DOFs and mesh entities, some equations or unknowns may be duplicated and **ghost equations** or **ghost unknowns** are equations or unknowns related to ghost mesh entities. In the linear system assembly phase, we must detect ghost equations to treat them in a special way and to ensure that the global linear system is square. With a local numbering mechanism that numbers local equations before ghost equations related to ghost entities, the local ids of local equations (respectively unknowns) are always lower than ghost equations (respectively unknowns). It is then easy to take into account or not the last equations which may not be valid as they are related to ghost entities.

parallelization principles

The principles of the parallelization is quite standard:

- each MPI process executes the different algorithms on its own data, and builds the local part of a distributed linear system;
- a parallel linear solver layer is used to solve the parallel linear system and to compute the solution of local DOFs;
- the values of ghost DOFs are updated with the Arcane synchronizer services.

Boundary conditions parallelization

The parallelization of algorithms related to boundary conditions is a little more tricky, as on each MPI subdomains we have to deal with both the physical boundaries and the MPI interfaces. In the assembly phase, we must only take into account closure equations related to physical boundaries and not to MPI interfaces. This is done using our numbering systems separating local entities from ghost ones.

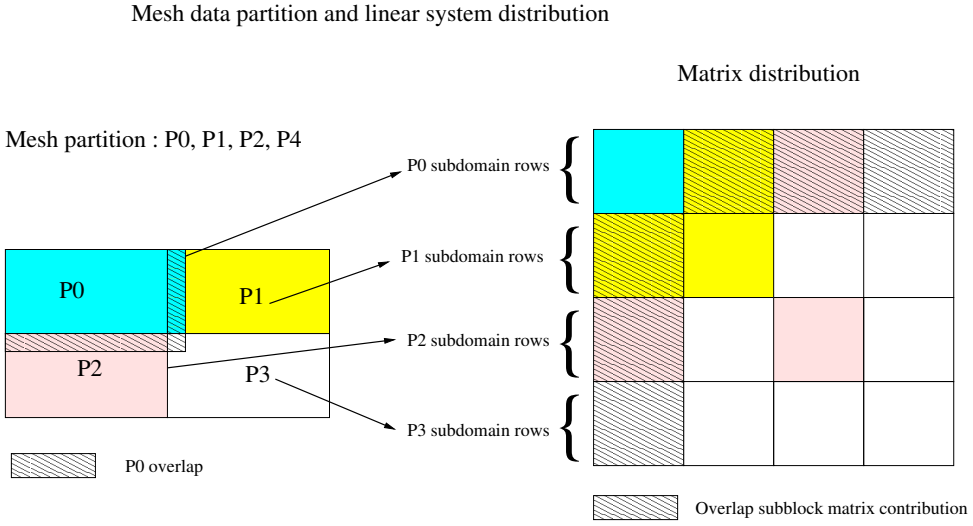


Figure 4.8: Mesh and matrix distribution on distributed memory architecture

Cell centered methods parallelization

For cell centered methods, there are only DOFs on cell entities. In many cases there are not boundaries closure equations, but implicit space constraints to take into account Dirichlet boundary conditions. The parallelization of such methods is more difficult since we cannot take into account such conditions on MPI interfaces. For example for the ccG method, the $\mathfrak{G}_h$ operator takes implicitly into account the values of functions on face boundaries. Let us consider a MPI subdomain and a face F of the MPI interface. This face has a back cell τ^b and a front cell τ^f. Let us suppose that the τ^b belongs to the subdomain. Then τ^f belongs to another MPI subdomain and not to the current subdomain. For τ^b, the gradient reconstruction operator is incomplete as we do not have the dirichlet boundary condition on F and we do not have access to the DOFs related to τ^f. The operator $\mathfrak{R}_h$ based on the operator $\mathfrak{G}_h$ with the green formula is also incomplete. We have a parallelization difficulty as the discretization of the jump operator on a local face connected to τ^b involves the $\mathfrak{R}_h$ operator on τ^b which is incomplete.

A solution to overcome this difficulty would be to add an extra ghost layer so that we would have access to the DOFs of τ^f. In figure 4.9, we consider two MPI domains, the face `F` belonging to the frontier of the domains and the back and front cells `K` and `L` of `F`. We can see that, to compute the **jump** operator on `F`, we need to evaluate the operator $\mathfrak{G}_h$ on `L` which leads to a linear combination with the stencil `stencilGradL` composed of the cells `L`, `L1`, `L2` and `L3`. We can understand then the necessity to add a second ghost layer to access to the DOFs related to `L2`. This solution gives poor performance for a large number of nodes because it increases a lot, the amount of communication when the number of MPI processes increases and the size of MPI subdomain decreases: this amount of communication is proportional to the size of the MPI interface boundary which gets fatter relatively to the local size of the MPI subdomain.

A better solution, more complex but more scalable, consists in introducing DOFs on MPI interface faces so that the $\mathfrak{G}_h$ operator remains valid even on ghost cells, and in adding extra closure equations on the MPI subdomains where these faces are local, using for example the trace operator $\mathfrak{T}_h$ which is complete as the face is local. For example in figure 4.9, introducing DOFs on the face `F2`, the linear combination of the operator $\mathfrak{G}_h$ evaluated on `L` is composed of `L1`, `L3` and the face `F2` which already belongs to the first ghost layer. We need to add an extra equation on the MPI Domain 2 to evaluate the DOFs on `F2`, which can be done with a trace operator.

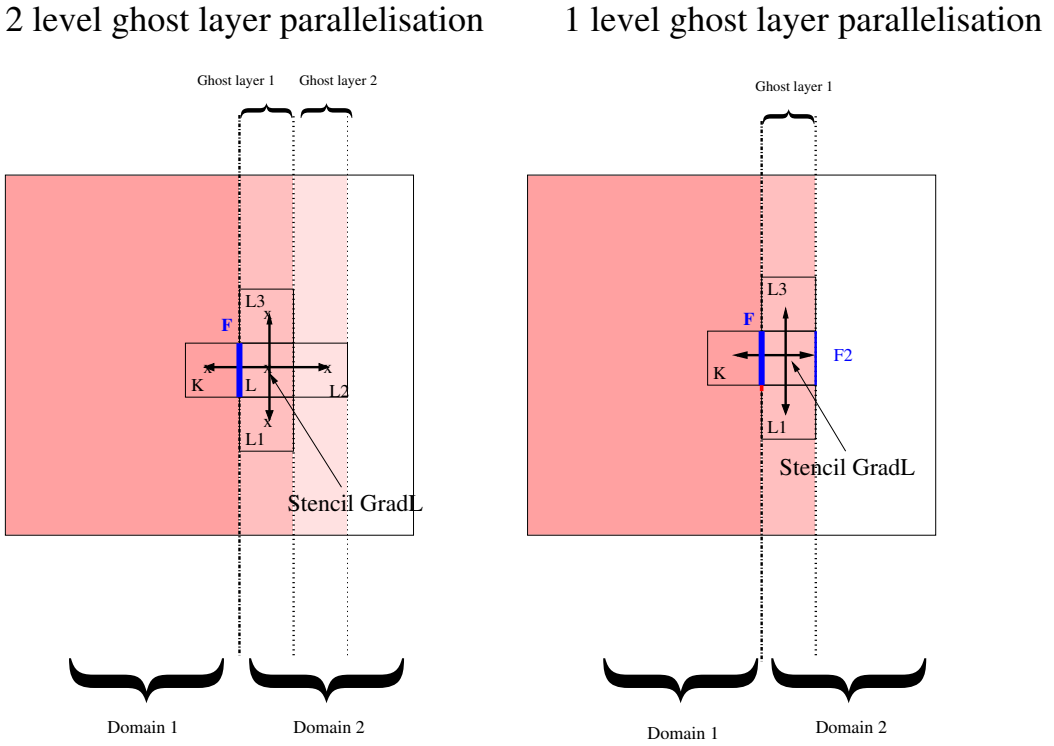


Figure 4.9: Cell centered methods parallelization

4.3.2 Task parallelization and granularity consideration

In modern architectures, within a node, we can have access to different kinds of computational units: processors with several cores and accelerator boards. To maximize the occupancy of all these computation units, we have to parallelize algorithms considering for each of them the different levels of parallelism and grain sizes. Let us consider for example the basis computation of the multiscale algorithm described in §2.5. Let N_f and N_c be the size of the fine and coarse grid, $R = \frac{N_f}{N_c}$ the rate of refinement. The number N_p of independent basis problems to solve is equal to the number of inner coarse faces,$N_p \approx \alpha_1 * N_c = \alpha_1 * \frac{N_f}{\eta}$ where α_1 is a factor dependent on the mesh dimension. The size N_b of each problem is proportional to the number of entities of each basis domain which is the number of fine entities of the back and front coarse cell of a coarse face. We have then $N_b \approx \alpha_2 * R$ where α_2 is another factor only depending on the mesh dimension. For a given fine problem size N_f, we can see that N_p and N_b are inversely proportional. The more independent problems there are, the smaller is each problem. This is a typical granularity size problem of parallelism which it is important to take into consideration to choose the best strategy of parallelism with respect to the type of available computation units. When η increases, N_p gets smaller and N_b larger, then we have a small number of big tasks. Such a configuration is well adapted to multi core technology with a small number of cores. The overhead of the construction and destruction of tasks can be balanced by the computation cost of each task. When η decreases, N_p gets large and N_b small, then we can have a great number of small tasks. Such a configuration requires solutions adapted to thin grain size parallelism that minimize the overhead of task construction and destruction. The ForkJoin (§4.2.3) and Pipeline (§4.2.3) concepts are useful tools to handle such a thin grain parallelism. When accelerator devices like GPGPU are available a thinner level of parallelism can be considered when the algorithm of each independent problem can be parallelized. For example, on GP-GPU devices, in our example, the algorithms of linear system resolution and matrix vector product can be parallelized. We can consider nested loops of parallelism or a pipeline mechanism to improve the rate of occupancy of

the GPU cores. The **GPUAlgebra framework** presented in §4.3.3 is a useful layer that helps to compute efficiently on GPGPU when we have linear algebra operations to perform on a great amount of linear systems (matrices and vectors) too small to be considered individually. Nowadays with standard GPU libraries, direct linear solvers are efficient on dense matrices of size greater than 5000 rows [32]. We consider that matrices with lower than 1000 rows are small. This layer performs linear algebra operations on a flow of small systems. This flow is processed in streams, each stream executed by a flow of instructions structured in a pipeline process.

4.3.3 GPUAlgebra framework

GPUAlgebra framework is a software layer aimed at performing linear algebra operations (matrix-vector products, linear system resolutions, linear vector operations) on a flow of independent matrices and vectors, efficiently on GP-GPUs. The originality of that framework is to exploit both the coarse level of parallelism due to the independency of the linear systems and the thinner level of the parallelism inside the algorithms of the linear operations. This two levels approach enables to have performance on a great number of linear systems even if the size of each system is not large enough to maximize the occupancy of the GPU device.

Context and Motivation

Over the last few years, several libraries have been developed to solve linear algebra problems on GPUs: The BLAS and LAPACK routines have been implemented for GPUs in libraries like CUBLAS and CULA [2, 3] by NVidia or in the MAGMA library [32] by teams of the University of Tennessee. Up to now, the effort has been essentially done for dense matrices. Some works concerning sparse matrices are mentioned in recent papers[39, 40]. The CULA library functionalities have been extended to sparse structures only recently. Nowadays most of these libraries are competitive with respect to standard CPU libraries (Lapack, ScaLapack, MKL) only for matrices with more than 5000 rows, generally for dense matrices and even more for sparse structures. The problematics of solving a great amount of sparse systems with less than 1000 rows is very particular and there is no existing efficient solution for GPUs.

Proposition

The efficiency of GPUAlgebraFramework to perform linear algebra operations on a collection of linear systems relies on:

- the design of structures representing the linear system data on GPU adapted to the algorithm of the linear operations;
- the optimization of the occupancy of the cores of the GPUs using the maximum of levels of parallelism;
- the reduction of the cost and the overhead of the data transfer between the main memory and the GPUs local memory.

In the first version of the framework, we focus on the implementation of the LU direct solver and the matrix-vector product operation. The parallelization of the LU factorization at a thin grain size is an important issue because it enables then to consider another level of parallelization, at a coarse grain size, based on the independency of the linear systems to solve. For such a parallelization, we have studied different matrix storage formats [89], to choose the one which preserves the sparsity of the matrix structure while enabling the parallelization of the LU factorisation algorithm with efficient parallel loop without indirections. Different techniques have also been considered to reduce the cost of the data transfer between the main memory and the GPU local memory as its overhead can reduce dramatically the interest of the computational power of GPUs. These techniques consist mainly in reducing the amount of data to transfer, in using the asynchronism to enable CPU computation overlapping and in using pinned memory mechanisms to enhance memory copy between host and devices.

Elements of implementation

In this section, we detail some elements of the LU factorisation implementation that enables us to improve the performance of the framework on GP-GPU devices. Our implementation depends on different hardware parameters: the `warp_size`, `max_block_size`, the `stack_size` and the number of streams `nb_stream`. These parameters are dynamic properties of our architecture model 4.2.2 initialized at the beginning of the application execution. Our framework has been designed mainly on NVidia GP-GPU device (Tesla S1070, Tesla C2070) with a certain number of streaming multi processors. On this kind of architecture, the parallelism is based on the concept of thread blocks, split in groups of threads WARP with a minimum number of threads that execute SIMD instructions. The `warp_size` and `max_block_size` parameters are used to evaluate the granularity of the thinner level of parallelism. Maximizing the occupancy of the device consists in ensuring that most of the streaming multi-processors are active. A multi-processor is inactive when there are not enough WARP ready to be executed. A WARP is not ready when it is waiting for a synchronization or waiting for a global memory request. When a kernel is executed, a thread block is either active, affected to a stream processor, or inactive, waiting for an available stream processor. The availability of a stream processor depends on the following characteristics:

- the maximum number of active blocks;
- the number of registries used by a thread;
- the shared memory size allocated by a block.

There are different ways to optimize stream processors occupancy:

- The first way is to ensure that most of the executed warps are full;
- On recent NVidia device, concurrent kernels can be executed in different streams. A second way consists in overlapping the latency, providing enough work in different streams. Thus when warps are waiting for synchronisation or waiting for a global memory request, there still remains other warps ready to be executed, on which stream processors could switch instead of remaining in an inactive state.

Parallelization principle The LU algorithm is parallelized with a classic approach based on a band storage column (BSC) matrix structure format. In the algorithm (listing 4.12) at each iteration of the outer loop on the row index `k`, we parallelize the inner loops on the row index `i` and column index `j` with a group of threads. The BSC format ensures that the two inner loops have a maximum size equal to the matrices `bandwidth` parameter.

Listing 4.12: standard LU factorization

```
DO k = 1, n-1
  DO i = k+1, n                          ! Column
    A(i, k) = A(i, k) / A(k, k)          ! Normalization
  END DO                                 !
  DO i = k+1, n                                  !
    DO j = k+1, n                                ! Submatrix
      A(i, j) = A(i, j) - A(i, k)*A(k, j)        ! Modification
    END DO                                       !
  END DO                                         !
END DO
```

When `bandwidth>warp_size`, the two inner loops are split in blocks of `warp_size`. Another outer loop on these blocks is introduced between the outer loop on `k` and the parallelized inner loops on `i` and `j`.

Otherwise, we consider the `rate=max_warp_size/bandwidth` that evaluates the number of independent systems that can be solved in parallel by a same group of threads. If this rate

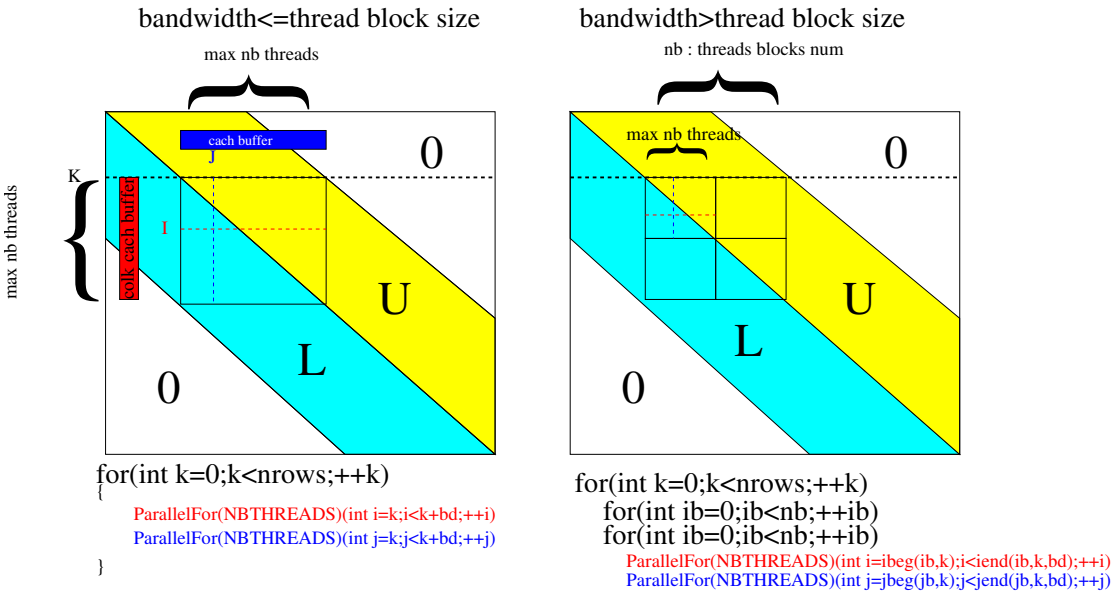


Figure 4.10: Parallel LU Factorisation on GPGPU

is big enough, we introduce another coarser level of parallelism to solve a number `nb` of matrices with the same `bandwidth`. This flow of systems is solved by a group of threads of size `nb*bandwidth<max_warp_size`.

We introduce two local cache buffers of size `nb*bande_size`

Data transfer The transfer of linear system data between the host memory and the local memory is executed asynchronously (figure 4.11). Linear systems are built on the host memory with a compressed storage row (CSR) format. Data is transferred in the local memory of the device, then a transformation from CSR to BSC format is executed asynchronously. These operations are executed within a stream that creates a pool of systems to solve on the device. The size of the pool is limited by a `pool_max_size` parameter depending on a device memory parameter. The resolution of the linear systems of a pool is executed when the `pool_max_size` parameter is reached or when all the built systems are transferred.

Stream feature As recent Nvidia cards support concurrent kernel execution, we have introduced the possiblity to create multiple streams managing different pools of linear systems. This feature has two advantages:

- it enables to manage the concurrency of multiple MPI processes sharing GPU devices in multi-core nodes when the number of GPUs is lower than the number of cores;
- it enables to improve the occupancy of each GPU by overlapping memory access or synchronization latency by the execution of concurrent kernels of multiple streams.

The figure 4.12 illustrates the flow of executions on a GPU device shared by two CPUs executing two different streams.

Matrix bandwidth management Matrices are stored on the device memory in a BSC format and the amount of memory used directly depends on the matrix bandwidth. We provide the Cuthill-Mckee renumbering algorithm that reduces the matrix bandwidth. Such a bandwidth reduction has two advantages:

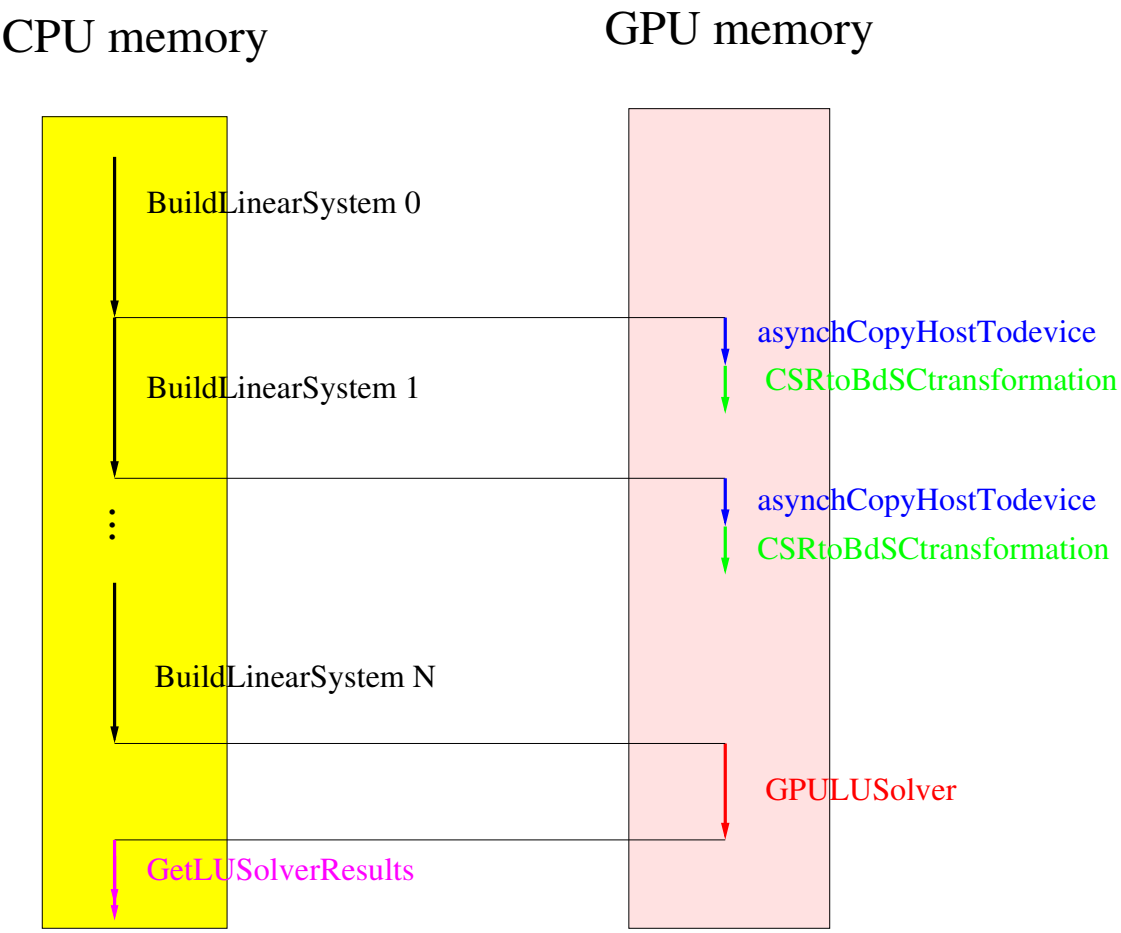


Figure 4.11: CPU and GPU memory management

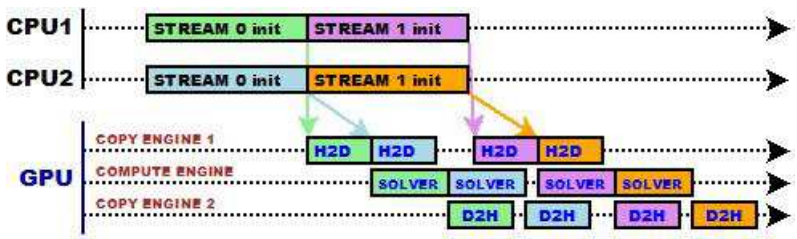


Figure 4.12: Streams and kernel concurrency

- it reduces the memory needed to store each system, increasing in that way the number of systems that can be managed by each system pool;
- it enables to increase the number of systems that can be solved by each group of threads.

4.4 Application to multiscale basis functions construction

We have validated the RunTime System Model presented in §4.2 implementing the basis function computation of the multiscale method. The purpose was to have a generic way to implement these computations for various hardware configurations and various implementations of the runtime system using muti-thread technology with TBB, Boost.Thread or pThread library for multi-core platform or with the GPUAlgebraFramework layer written with Cuda or OpenCL for node with GP-GPU accelerators.

We have implemented a `BasisFunction` class with a standard implementation for CPU and a GPU implementation based on the GPUAlgebraFramework layer for GP-GPU devices. The algorithm to compute all the basis functions has been written as in listing 4.13 with the `Task`, `ForkJoin` and `Pipeline` concept, and with various fork-join driver implementations based on the TBB, Boost.Thread, pThread and with the pipeline driver based on the GPUAlgebraFramework library.

Listing 4.13: Basis computation algorithm

```
   template<typename SchedulerT,
            typename TaskMngT,
            typename ForkJoinDriverT,
            typename PipelineDriverT,
            typename DataMngT>
void computeBasis(std::vector<BasisFunction*>& basis)
{
  typedef typename TaskMngT::uid_type                          uid_type ;
  typedef typename TaskMngT::ForkJoinTask<ForkJoinDriverT> ForkJoinTask ;
  typedef typename TaskMngT::PipelineTask<PipelineDriverT> PipelineTask ;
  typedef typename TaskMngT::TaskNode                          TaskNode ;
  typedef typename TaskMngT::Task<Basis>                       TaskType ;

  //DATA MANAGEMENT
  DataMng data_mng ;
  DataHandlerType* solver_handler = data_mng.getNewData() ;
  solver_handler->set<ILinearSolver>(basis_solver) ;
  DataHandlerType* k_handler = data_mng.getNewData() ;
  k_handler->set<VariableCellReal const>(&k) ;

  //TASK MANAGEMENT
  std::vector< uid_type >  dag ;
  TaskMng task_mng ;

  //DEFINE PARALLEL FORKJOIN FOR MULTICORE ARCHITECTURE
  ForkJoinDriverT forkjoin(/* ...*/) ;
  ForkJoinTask* forkjoin_task =
        new ForkJoinTask(forkjoin,task_mng.getTasks());
  uid_type fk_uid = m_task_mng.addNew(forkjoin_task) ;

  //DEFINE PIPELINE FOR GPU ARCHITECTURE
  PipelineDriverT pipeline(/* ...*/) ;
  PipelineTask* pipeline_task =
       new PipelineTask(pipeline,task_mng.getTasks()) ;
```

```
  uid_type pipeline_uid = m_task_mng.addNew(forkjoin_task) ;

  //DEFINE A DAG ROOT NODE WITH CPU AND GPU IMPL
  TaskNode* root = new TaskNode() ;
  root->set("cpu",fk_uid) ;
  root->set("gpu",pipeline_uid) ;
  uid_type root_uid = task_mng.addNew(root) ;

  //ADD ROOT TASK LIST AS A DAG ROOT NODE
  dag.push_back(root_uid) ;

  //DEFINE BASIS TASKS AND TASK DEPENDANCIES
  std::for_each(auto ibasis : basis)
  {
     TaskType* task = new TaskType(*ibasis) ;
     task->args().add("Solver",DataHandlerType::R,solver_handler) ;
     task->args().add("K",DataHandlerType::R,k_handler) ;
     typename TaskType::FuncType f_cpu = &BasisFunctionType::computeCPU ;
     task->set("cpu",f_cpu) ;
     task->set("gpu",f_gpu) ;
     Integer uid = m_task_mng.addNew(task) ;
     forkjoin_task->add(uid) ;
     pipeline_task->add(uid) ;
  }

  //EXECUTE THE DAG
  SchedulerT scheduler ;
  task_mng.run(scheduler,dag) ;
}
```

The parallelisation on hybrid architectures has two main levels:

- the first level is based on the parallelization of the coarse problem on the coarse grid with the method described in §4.3.1. The coarse mesh is partitioned and we create MPI coarse subdomains adding ghost elements. The distribution of the fine mesh follows the coarse mesh partition. MPI fine subdomains are built with the fine elements of the fine mesh belonging to coarse elements of the MPI coarse subdomain including ghost elements. On each MPI coarse subdomain the basis functions related to local coarse faces are computed. That can be done since the computation domain of the basis function are complete even if they are based on a ghost coarse cell. The parallelisation of the coarse method is then standard as all the assembly elements are local.
- On heterogeneous nodes with several cores with several accelerator boards, a second level of parallelization is done on the computation of the local basis function transparently with the runtime system as described previously. This parallelisation is based on the use of the `ForkJoin` concept for multi-core nodes and of the `Pipeline` concept for node enhanced with accelerator boards.

4.5 Performance results

In this section we present some performance results of the basis functions computation of the multiscale method implemented with our RunTime System Model on a benchmark of the 2D SPE10 study case described in §5.4.3. We compare different implementations and solutions run on various hardware configurations. We focus on the test case with a 65x220x1 fine mesh and a 10x10x1 coarse mesh which leads to solve 200 linear systems of approximately 1300 rows. We apply the reducing bandwidth renumbering algorithm to all matrices and their bandwidth is lower than 65 for all them.

4.5.1 Hardware descriptions

The benchmark test cases have been run on two servers (figure 4.13):

- the first one, Server 1 is a Bull novascale server with a SMP node 2 quad-core intel Xeon E5420 GPU tesla server S1070 with 4 GPU tesla T10 with 30 streaming processors with 8 cores, 240 computation units per processor, total of 960 for the server. 16 GB central memory;
- the second, Server 2 is a server with a SMP node with 2 octo-core processors Intel Xeon E5-2680 linked by a NUMA memory and with 2 GPUs Tesla C2070 per processor with a fermi architecture.

4.5.2 Benchmark metrics

In our benchmark we focus on the execution time in seconds of the computation of all the basis functions of the study case. This computation time includes for each basis function, the time to discretize the local PDE problem, to build the algebraic linear system, to solve it with a linear solver and to finalize the computation of the basis functions updating them with the solution of the linear system.

To analyze in detail the different implementations, we also separately measure in seconds:

- t_{start} the time to define basis matrix structures;
- $t_{compute}$ the time to compute the linear systems to solve;
- t_{sinit} the setup time of the solver;
- t_{solver} the time to solve all the linear systems;
- $t_{finalize}$ the time to get the linear solution and finalize the basis function computation;
- t_{basis} the global time to compute all the basis functions.

The performance results are organized in tables and graphics containing different times in seconds which can be compared to the time of a reference execution on one core.

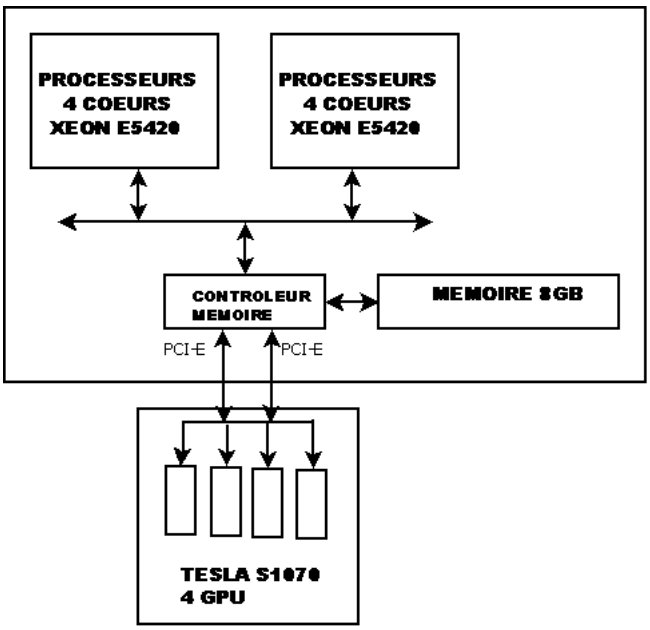
4.5.3 Results of various implementations executed on various hardware configurations

Multithread forkjoin and GPU pipeline implementation In table 4.14 and figure 4.15, we compare the performances of:

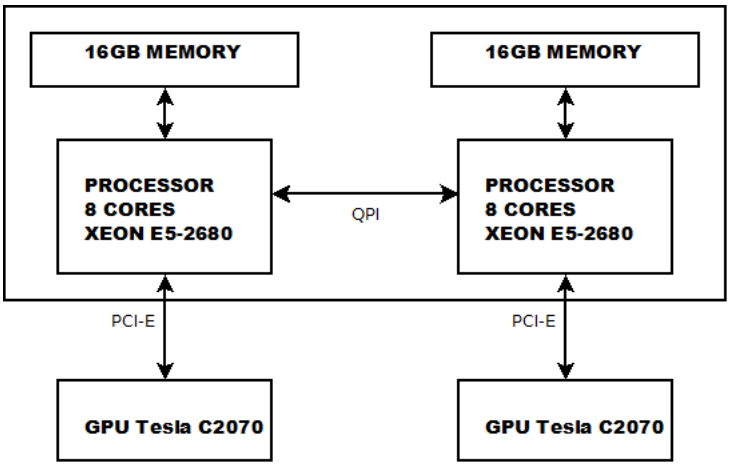
- the forkjoin concept implementations `TBB`, `BTH` and `PTH` using respectively the `TBBDriver`, `BTHDriver` and `PTHDriver` drivers which are all thread based implementations for multi-core configuration.
- various pipeline concept implementations based on the GPUAlgebraFramework using one or multiple CUDA streams.

We study the following hardware configurations:

- `cpu`, the reference configuration with 1 core;
- `gpu`, configuration with 1 core, 1 gpu;
- `sgpu-1`, configuration with 1 core, 1 gpu and 1 stream;



(a) Server 1



(b) Server 2

Figure 4.13: Servers architecture

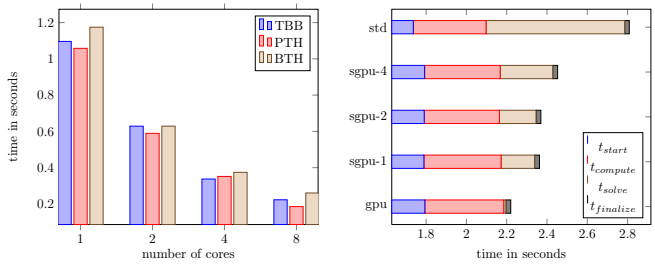
NbThreads	1	2	4	8	16
TBB	1.09	0.62	0.33	0.22	
Boost Thread	1.17	0.62	0.37	0.26	
Posix Thread	1.05	0.58	0.35	0.18	

(a) Basis functions computation time vs number of threads

opt	t_{start}	$t_{compute}$	t_{sinit}	t_{solver}	$t_{finalize}$	t_{basis}
cpu	1.73	0.36	0.	0.68	0.022	2.80
gpu	1.79	0.39	1.36	0.01	0.024	3.59
sgpu-1	1.78	0.38	1.29	0.16	0.024	3.67
sgpu-2	1.79	0.37	1.30	0.18	0.024	3.68
sgpu-4	1.79	0.37	1.31	0.26	0.024	3.77

(b) Basis computation phase time for various solver configuration

Figure 4.14: Server 2: Multi-thread and GPU implementation results



(a) TBB,PTH,BTH (b) GPU

Figure 4.15: Performance analysis for multi-core configuration and GPU configurations

- `sgpu-2`, configuration with 1 core, 1 gpu and 2 stream;
- `sgpu-4`, configuration with 1 core, 1 gpu and 4 stream;
- `n x p core`, configuration with n cpus and p cores per cpu.

In figure 4.15, we compare three implementations of the fork-join behaviour with threads. The analysis of the results shows that they all enable us to improve the efficiency of the basis function computation taking advantage of the multi-core architecture. The `PTH` implementation, directly written by hand with Posix threads is the most efficient while the `PTH` one implemented with Boost threads the less. The `TBB` version efficiency is between the two others. Comparing the various implementation of the pipeline behaviour for GPU, we can notice that only the solver part is really accelerated on the GPU. The influence of the number of cuda stream on the performance is not evident on that test case. Nevertheless all the GPU implementations enable to improve the efficiency of the basis function computation with respect to the standard version on one core. Finally all these results prove that we can handle various hardware architectures, with one or several cores, with or without several GPGPUs, with a unified code. That illustrates the capacity of the runtime system to hide the hardware complexity in a numerical algorithm.

Multi-core multi-GPU configuration For multi-core and multi-GPU configuration, we study the performance of a mixed MPI-GPU implementation with two levels of parallelism:

- the first level is a MPI based implementation for distributed memory;
- the second level is based on the `GPUAlgebraFramework` to solve the linear systems on GP-GPU devices.

ncpu	1 gpu	2 gpus	4 gpus
1	1.95	1.95	1.95
2	1.22	1.04	1.04
4	0.98	0.76	0.66
8	0.63	0.37	0.45

(a) Computation times vs number of cpus and gpus

ngpu	2 x 2 cores	1 x 4 cores	1 x 8 cores
1	1.06	1.05	0.51
2	0.76	0.76	0.37
4	0.66	0.66	0.40

(b) Computation times vs number of cpus, cores per cpu and gpus

Figure 4.16: Server 1: multi-cores multi-gpu configuration

ncpu	1 gpu	2 gpus
1	0.75	0.75
2	0.44	0.38
4	0.25	0.24
8	0.12	0.12
16	0.05	0.06

(a) Computation times vs number of cpus and gpus

ngpu	2 x 2 cores	1 x 4 cores	2 x 4 cores	1 x 8 cores
1	0.53	0.67	0.57	0.50
2	0.46	0.45	0.37	0.37

(b) Computation times vs number of cpus, cores per cpu and gpus

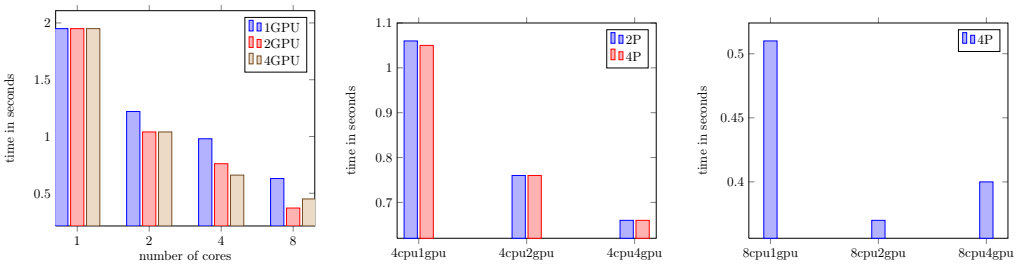
Figure 4.17: Server 2: multi-cores multi-gpu configuration

We test different hardware configurations with different number of cores (1,2,4,8 and 16) sharing 1, 2 or 4 GPUs. In table 4.16 and figure 4.18 (respectively table 4.17 and figure 4.19) we present the performance results for the server 1 (respectively server 2).

The results show that the runtime system enable us to easily compare various hardware configurations: configurations where gpus are shared or not by cpus and cores, configurations with different strategies of connexion between gpus and cpus.

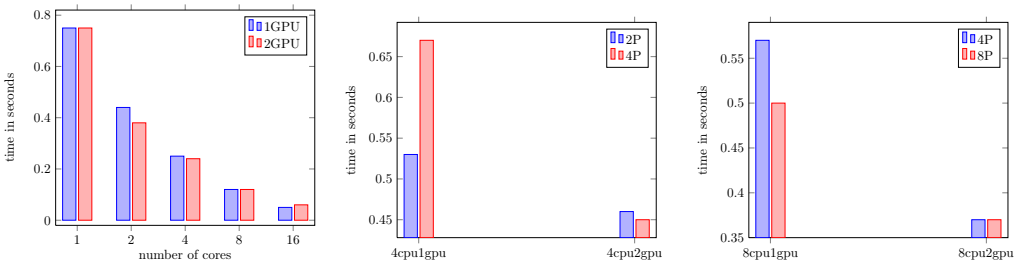
Conclusions Analyzing the results of the different benchmarks, we have different levels of conclusions:

- the first level concerns the capacity of the Runtime system to hide the hardware complexity in a numerical algorithm. These benchmarks prove that we can handle various hardware architectures, with one or several cores, with or without several GPGPUs, with a unified code.
- the second level concerns the extensibility of the Runtime system. We could compare competing technologies with different implementations of our abstract concepts with few impacts on the numerical code.



(a) Multi CPU, multi GPU (b) 4 CPU, multi GPUs (c) 8 CPU, multi GPUs

Figure 4.18: Server 1: multi-cores multi-gpu configuration



(a) Multi CPU, multi GPU (b) 4 CPU, multi GPUs (c) 8 CPU, multi GPUs

Figure 4.19: Server 2: multi-cores multi-gpu configuration

- the third level concerns the capability of the Runtime system to really improve the performance of the numerical algorithm using the different levels of parallelism provided by hybrid architecture. With all the technologies tested the performance of the computation has been improved compared to one computation executed on one core.
- the last level of conclusion is the fact that the runtime system enables to benchmark in a simply way the different hardware configurations parameters like the number of cores, the number of GPUs, the number of streams, the fact that a GPU is shared or not by several cores.

Perspective Our first results have prove the interest of our approach to handle the variety of hardware technology with few impacts on the numerical layer. Nevertheless the solutions we have implemented are still to simple to get the maximum of the performance that can provide new heterogeneous architectures. We need to implement our different abstractions with advanced solutions as those existing in research runtime system solutions like StarPU or XKaapi. We plan also to benchmark different mechanisms that help to optimize data transfer between main memory and local accelerator memories and to measure the overhead of each solution with respect to the parallelism grain sizes.

Chapter 5

Results on various applications

In this chapter we study various application problems. We present their mathematical continuous settings and we detail different variational discrete formulations that we compare to their programming counterpart with the DSEL. We evaluate the flexibility of the language to describe and to implement various discretization methods. We benchmark the study cases on different hardware configurations and present their performance results.

We study first two application problems, representative of the diffusive models solved in reservoir and CO2 storage simulation. We study then the incompressible Navier Stokes problem which is a more complex academic problem. We study finally a problem, inspired from the well known SPE10 reservoir benchmark with an heterogeneous data set, usually used as a representative difficult model of oil industry.

For these benchmarks, the prototypes are compiled with a gcc 4.5 compiler with the following compilation flags "-O3 -mssse3". The test cases are run on a server with a SMP node with 2 octo-core processors Intel Xeon E5-2680 linked by a NUMA memory and with 2 GPUs Tesla C2070 per processor with a fermi architecture.

5.1 Advection diffusion reaction

In this section we study an advection diffusion reaction problem modeling the injection of CO2 in a porous media domain.

5.1.1 Problem settings

Let $\Omega \subset \mathbb{R}^d$, $d \geq 2$, the mathematical coutinuous settings reads:

$$\begin{cases} \nabla\cdot(-\nu\nabla u + \beta u) + \mu u = f \text{ in } \Omega, \\ u = g \text{ on } \partial\Omega, \end{cases} \tag{5.1}$$

with $\nu > 0$, $\beta \in \mathbb{R}^d$, $\mu \geq 0$, $f \in L^2(\Omega)$ and $g \in L^2(\partial\Omega)$.

The continuous weak formulation reads:
Find $u \in H_0^1(\Omega)$ such that

$$a(u,v) = b(v) \quad \forall v \in H_0^1(\Omega), \tag{5.2}$$

with

$$\begin{cases} a(u,v) \stackrel{\text{def}}{=} \int_\Omega -\nu\nabla u\cdot\nabla v + (\beta\cdot\nabla u)v + \mu uv, \\ b(v) \stackrel{\text{def}}{=} \int_\Omega fv \end{cases} \tag{5.3}$$

The discretization of the variational formulation with the ccG method presented in 2.4.2 reads: Let $\mathcal{T}_h$ a mesh representation of Ω, $U_h(\mathcal{T}_h)$ a ccG space, find $u_h \in U_h(\mathcal{T}_h)$ such that:

$$a_h(u_h, v_h) = b_h(v) \quad \forall v_h \in U_h(\mathcal{T}_h) \tag{5.4}$$

with

$$\begin{aligned} a_h(u_h, v_h) \overset{\text{def}}{=} & \int_\Omega -\nu\nabla u_h \cdot \nabla v_h + (\beta \cdot \nabla u_h)v_h + \mu u_h v_h) \\ & + \sum_{F_h \in \Omega_h} (\int_{F_h} -\nu[\![u_h]\!](\{\nabla u_h\} \cdot \mathrm{n}_{F_h}) - \nu(\{\nabla u_h\} \cdot \mathrm{n}_{F_h})[\![v_h]\!] \\ & \qquad + (\frac{\eta}{h} + \frac{1}{2}|\beta \cdot \mathrm{n}|)[\![u]\!][\![v]\!] - (\beta \cdot n)[\![u_h]\!]\{v_h\}) \\ & + \sum_{F_h \in \partial\Omega_h} (\int_{F_h} -\nu[\![u_h]\!](\{\nabla v_h\} \cdot \mathrm{n}_{F_h}) - \nu(\{\nabla u_h\} \cdot \mathrm{n}_{F_h})[\![v]\!] \\ & \qquad + (\frac{\eta}{h} + \frac{1}{2}|\beta \cdot \mathrm{n}|)[\![u_h]\!][\![v_h]\!] + (\beta \cdot n)^- u_h v_h) \\ b_h(v_h) \overset{\text{def}}{=} & \int_\Omega f v_h \end{aligned} \tag{5.5}$$

This formulation can be compared to its programming counterpart in listing 5.1.

Listing 5.1: C++ implementation of the diffusion-advection-reactive problem

```
MeshType Th ;          // declare Th
Real nu, mu, eta ; // declare ν, μ and η
auto Uh = newCCGSpaceType(Th) ;
auto u = *Uh->trial("U") ;
auto v = *Uh->test("V") ;
auto lambda1 = eta/H() + 0.5*abs(dot(beta,N())) ;
auto lambda2 = eta/H() + 0.5*abs(dot(beta,N())) ;
BilinearForm ah =
 integrate( allCells(Th),
            dot(nu*grad(u),grad(v)) +
            dot(beta,grad(u))* v +
            mu* u* v ) +
  integrate( internalFaces(Th),
            -nu*jump(u)*dot(N(),avr(grad(v))) -
            nu*dot(N(),avr(grad(u)))*jump(v) +
            lambda1*jump(u)*jump(v) -
            dot(beta,N())*jump(u) *avr(v) ) +
  integrate( boundaryFaces(Th),
            -nu*jump(u)*dot(N(),avr(grad(v))) -
             nu*dot(N(),avr(grad(u)))*jump(v) +
             lambda2*jump(u)*jump(v) +
             ominus(dot(beta,N()))* u * v) ;

LinearForm bh =
  integrate( allCells(Th),val(m_f)*id(v) ) +
  integrate( boundaryFaces(Th),
            ominus(dot(beta,N())* g) * v ) ;
```

5.1.2 Results

We consider the analytical solution of the advection diffusion problem (5.1) on the square domain $\Omega = [0,1]^2$ with $\eta = 1$, $\mu = 0$. $\beta = (1., 0., 0.)$, $f(x,y) = 2sin(x)(cos(x) + 2sin(y))$ and

$g(x, y) = sin(x)sin(y)$.

The test case is run on a family of meshes of increasing sizes. We evaluate the errors to the analytical solution, with the following norms:

- $\|u\|_{L^2}^2 = \int_\Omega u^2$;
- $\|u\|_L^2 = \sum_{T\in\mathcal{T}_h} \sum_{F\in\mathcal{F}_T} \int_F \frac{1}{d_{F,T}^2} \mathfrak{T}(u)^2$;
- $\|u\|_F^2 = \sum_{F\in\mathcal{F}_h} \int_F \mathfrak{T}(u)^2$;
- $\|u\|_G^2 = \sum_{F\in\mathcal{F}_h} \int_F \mathfrak{T}(u)^2 + \int_\Omega \|\nabla u\|^2$.

and estimate the order of convergence as

$$\text{order} = d \ln\left(e_1/e_2\right) / \ln\left(\text{card}(\mathcal{T}_{h_2})/\text{card}(\mathcal{T}_{h_1})\right),$$

where e_1 and e_2 denote, respectively, the errors committed on $\mathcal{T}_{h_1}$ and $\mathcal{T}_{h_2}$, h_1, $h_2 \in \mathcal{H}$.

To analyze the performance of the framework, we evaluate the overhead of the language, the relative part of algebraic computations (defining, building and solving linear systems) and linear combination computations by monitoring the following times:

- t_{start} the time to precompute trace and gradient operators, to build the expression tree describing linear and bilinear forms;
- t_{def} the time to compute the linear system profile;
- t_{build} the time to build the linear system evaluating the expression tree;
- t_{solve} the time to solve le linear system with linear algebra layer;
- N_{it} the number of iterations of the linear solver, a ILU0 preconditioned BiCGStab algorithm with relative tolerance set to 1.10^{-6};
- N_{nz} the number on non zero entries of the linear system of the test case.

All these times in seconds are compared to $t_{ref} = \frac{t_{solver}}{N_{it}}$ the solver time per iteration which is equivalent to a fixed number of matrix vector multiplication operations. This reference time is a good candidate to compare the cost of each part of the computation as it enables comparisons to other numerical methods, for a given numerical error.

In iterative methods (time integration, non linear solver), t_{start} and t_{def} correspond to computation phases often factorized and done once before the first iterative step, while the t_{build} corresponds to a computation phase done at each steps. A careful attention has to be paid to the t_{build} results specially for iterative algorithms.

To evaluate the memory consumption we monitor Nnz, the number of non zero entries of the linear system.

Figure 5.1 is a 2D view of the solution. Convergence results are listed in Table 5.1. Performance results are listed in table 5.2.

The analysis of these results shows that the ccG-method has the expected convergence described in [49]. The implementation remains scalable with respect to the mesh size.

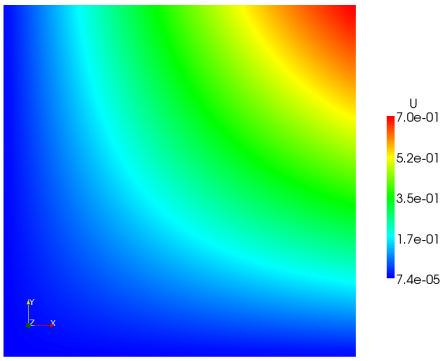
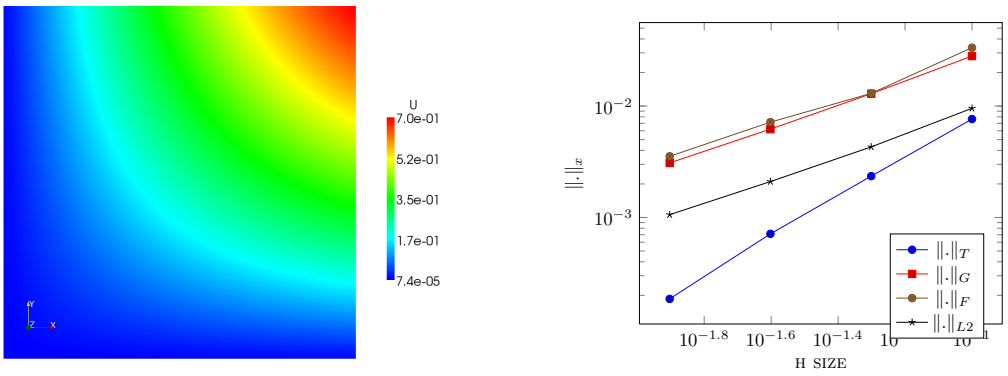


Figure 5.1: Advection diffusion solution



(a) 2D view (b) convergence curves

Figure 5.2: Advection Diffusion reaction problem

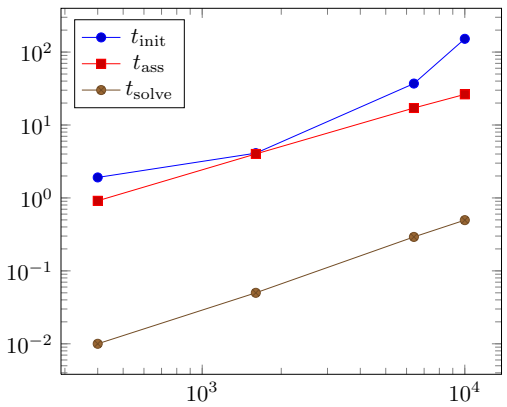


Figure 5.3: Performance analysis for the example (5.1)

Table 5.1: Advection diffusion test case

card($\mathcal{T}_h$)	h	$\|u-u_h\|_T$	order	$\|u-u_h\|_G$	order	$\|u-u_h\|_F$	order	$\|u-u_h\|_{L2}$	order.
400	1.00E-01	7.64E-03		2.81E-02		3.34E-02		9.54E-03	
1600	5.00E-02	2.35E-03	1.7	1.29E-02	1.12	1.30E-03	1.12	4.30E-03	1.15
6400	2.50E-02	7.13E-04	1.71	6.21E-03	1.09	7.17E-03	1.11	2.10E-03	1.09
25600	1.25E-02	1.86E-04	1.79	3.09E-03	1.06	3.54E-03	1.08	1.06E-03	1.06

Table 5.2: Advection diffusion test case: performance results

card($\mathcal{T}_h$)	N_{it}	N_{nz}	t_{start}	t_{def}	t_{build}	t_{solve}	t_{ref}	$\frac{t_{start}}{t_{ref}}$	$\frac{t_{build}}{t_{ref}}$
400	3	35734	1.91E00	9.00E-01	9.1E-01	1.00E-02	3.33E-03	274.21	270.31
1600	5	165638	4.11E00	4.11E00	4.01E00	5.00E-02	1.00E-02	400.6	411.39
6400	8	723906	3.70E01	1.70E01	1.71E01	2.92E-01	3.65E-02	469.7	466.74
25600	10	1120118	1.52E01	2.70E01	2.63E01	4.96E-01	4.96E-02	531.07	545.18

5.2 SHPCO2 test case

In this section we study a synthetic test case proposed in the ANR project SHPCO2, inspired from one of the benchmarks proposed by the GDR MoMaS [4] 2008 integrating specific elements of the CO2 problematics.

5.2.1 Problem settings

We consider the following heterogeneous diffusion model problem :

$$\begin{aligned} -\nabla\cdot(\kappa\nabla u) &= 0 \quad \text{in } \Omega, \\ u &= g \quad \text{on } \partial\Omega_d, \\ \partial_n u &= 0 \quad \text{on } \partial\Omega_n \end{aligned} \tag{5.6}$$

where:

- $\Omega \subset \mathbb{R}^2$ is described in figure 5.4. The problem size depends on the characteritic length $L = 1000m$. The depht of the 3D volum is $H = 100m$.
  The domain is partioned into two parts:
  - the drain part with a high permeability tensor;
  - the barrier part (in green) with a low permeability tensor.

- The domain boundary $\partial\Omega$ is partitioned in 4 specific areas:
  - $\partial\Omega_{Inj1}$ and $\partial\Omega_{Inj2}$ with an injector imposed pressure boundary conditions $Pinj_1$ and $Pinj_2$,
  - $\partial\Omega_{prod}$ with a productor imposed pressure boundary condition $Pprod$,
  - and the boundary complementary part $\partial\Omega_n$ with an homogen Neumann boundary condition (null flux).

  Ω_d corresponds to $\partial\Omega_{Inj1} \cup \partial\Omega_{Inj2} \cup \partial\Omega_{Prod}$;

- g is equal to $Pinj_1$, $Pinj_2$ and $Pprod$ on respectively $\partial\Omega_{Inj1}$, $\partial\Omega_{Inj2}$ and $\partial\Omega_{Prod}$;

- κ is a permeability tensor with K_{drain} value on the drain part and $K_{barrier}$ value on the barrier part.

The problem (5.6) has been discretized with the SUSHI-method defined in Chapter 2 and implemented as in listing 5.2

Listing 5.2: C++ implementation of a_h^{hyb}

```
MeshType Th; // declare T_h
auto Uh = newSUSHISpace(Th);
auto u = Uh->trial("U");
auto v = Uh->test("V");
BilinearForm ah = integrate( allCells(Th),k*dot(grad(u),grad(v)) );
LinearForm bh = integrate( allCells(Th),f*v ) ;

//Dirichlet boundary condition
ah += on(boundaryFaces(Th,"dirichlet"),trace(u)=g );

//Neunman boundary condition
bh += integrate(boundaryFaces(Th,"neumann"),h*trace(u));
```

5.2.2 Results

Simulation parameters The problem has been solved with the following parameters:

- $\Phi = 0.2$
- $Pinj_1 = 110.e^5 Pa$,
- $Pinj_2 = 1.5.e^5 Pa$,
- $Pprod = 100.e^5 Pa$

Figure 5.4 illustrates the permeability field with the following parameters:

- $K_{drain} = 100.e^{-15} m^2$
- $K_{barrier} = 1.e^{-15} m^2$

Figure 5.4 gives a 2D view of the problem solution.

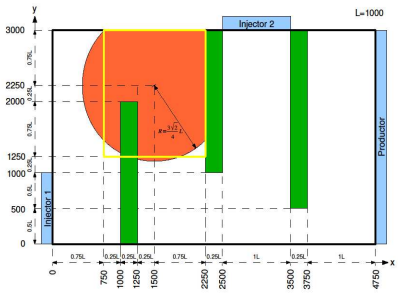
Performance results The study case is run on 1, 2, 4 and 8 cores. We collect t_{init}, t_{buid} and t_{solve} respectively the times in seconds to initialize the problem, to build the linear system and to solve it in table 5.2.2 and in the graphic of figure 5.2.2. These results show that the implementation of the t_{init}, t_{build} and t_{solve} phases is scalable and their relative costs between each others remain constant.

5.3 Navier-Stokes

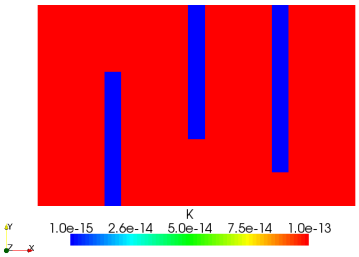
5.3.1 Problem settings

We consider the steady incompressible Navier-Stokes problem:

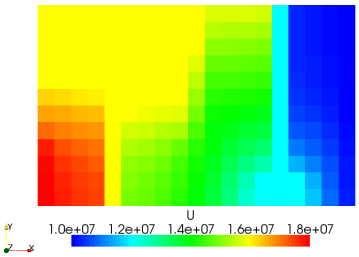
$$\begin{aligned} -\nu\triangle\mathbf{u} + \mathbf{u}\cdot\nabla\mathbf{u} + \nabla p &= \mathbf{f} \text{ in } \Omega, \\ \nabla\cdot\mathbf{u} &= 0 \text{ in } \Omega, \\ \mathbf{u} &= \mathbf{g} \text{ on } \partial\Omega, \\ \int_\Omega p &= 0, \end{aligned} \tag{5.7}$$



(a) Domain description



(b) Permeability field



(c) Pressure solution

Figure 5.4: SHPCO2 problem

NCPU	1	2	4	8
t_{init}	1.09	0.62	0.33	0.22
t_{build}	1.17	0.62	0.37	0.26
t_{solve}	1.05	0.58	0.35	0.18

Figure 5.5: SHPCO2: Performance results of the Hybrid method

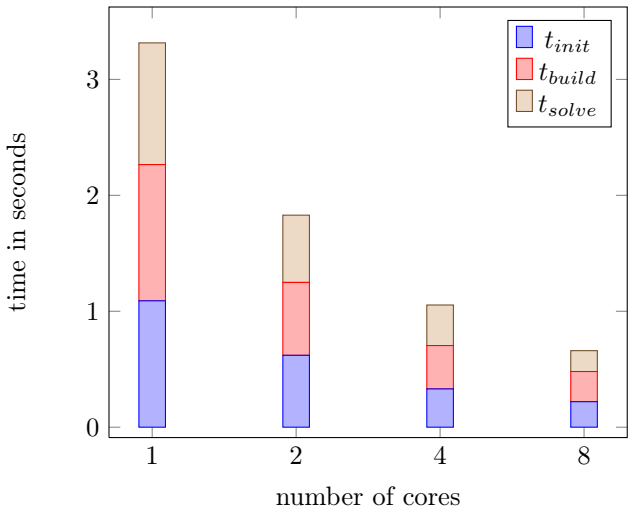


Figure 5.6: SHPCO2: Performance results of the Hybrid method

with $\Omega \subset \mathbb{R}^d$, $\mathbf{u} : \Omega \to \mathbb{R}^d$ and $p : \Omega \to \mathbb{R}$.

The continuous weak formulation for $g = 0$ (when $g \neq 0$ a trace lifting must be considered) reads:

Find $(\mathbf{u}, p) \in [H_0^1(\Omega)]^d \times L_*(\Omega)$ such that

$$a(\mathbf{u}, \mathbf{v}) + b(p, \mathbf{v}) - b(q, \mathbf{u}) + t_h(\mathbf{u_h^n}, \mathbf{u_h}, \mathbf{v_h}) = \int_\Omega f \cdot \mathbf{v} \quad \forall (\mathbf{v}, q) \in [H_0^1(\Omega)]^d \times L_*(\Omega),$$

with

$$a(\mathbf{u}, \mathbf{v}) \stackrel{\text{def}}{=} \int_\Omega \nabla \mathbf{u} : \nabla \mathbf{v}, \quad b(q, \mathbf{v}) \stackrel{\text{def}}{=} -\int_\Omega \nabla q \cdot \mathbf{v} = \int_\Omega q \nabla \cdot v.$$

The discretization of the variational formulation with the ccG method detailed in [49] reads:

Let $\mathcal{T}_h$ a mesh representation of Ω, $U_h \stackrel{\text{def}}{=} [V_h^{\text{ccg}}]^d$, $P_h \stackrel{\text{def}}{=} \mathbb{P}_d^0(\mathcal{T}_h)/\mathbb{R}$, find $(u_h, p_h) \in U_h \times P_h$ such that, for all $(v_h, q_h) \in U_h \times P_h$,

$$a_h(u_h, v_h) + b_h(v_h, p_h) - b_h(u_h, q_h) + t_h(u_h, u_h, v_h) = \int_\Omega f \cdot v_h. \tag{5.8}$$

where,

$$\begin{aligned} a_h(u_h, v_h) \stackrel{\text{def}}{=} \int_\Omega & -\nu \nabla u_h \cdot \nabla v_h \\ & + \sum_{F_h \in \Omega_h} \int_{F_h} -\nu [\![u_h]\!](\{\nabla u_h\} \cdot \mathrm{n}_{F_h}) - \nu(\{\nabla u_h\} \cdot \mathrm{n}_{F_h})[\![v_h]\!] + \\ & \sum_{F_h \in \partial\Omega_h} \int_{F_h} -\nu [\![u_h]\!](\{\nabla v_h\} \cdot \mathrm{n}_{F_h}) - \nu(\{\nabla u_h\} \cdot \mathrm{n}_{F_h})[\![v]\!] \end{aligned} \tag{5.9}$$

$$b_h(p_h, v_h) \stackrel{\text{def}}{=} \sum_{T_h \in \mathcal{T}_h} \int_{T_h} -p_h \nabla \cdot v_h + \sum_{F_h \in \mathcal{F}_h} \int_{F_h} \{p_h\}(\mathrm{n}_{F_h} \cdot [\![v_h]\!])) \tag{5.10}$$

$$b_h(p_h, v_h) \stackrel{\text{def}}{=} \sum_{T_h \in \mathcal{T}_h} \int_{T_h} \nabla p_h \cdot v_h - \sum_{F_h \in \mathcal{F}_h^i} \int_{F_h} [\![p_h]\!](\mathrm{n}_{F_h} \cdot \{v_h\}) \tag{5.11}$$

$$\begin{aligned} t_h(w_h, u_h, v_h) \stackrel{\text{def}}{=} & \sum_{T_h \in \mathcal{T}_h} \int_{T_h} (w_h \cdot \nabla u_{h,i}) v_{h,i} - \sum_{F_h \in \mathcal{F}_h^i} \int_{F_h} (\{w_h\} \cdot \mathrm{n}_{F_h})([\![u_h]\!] \cdot \{v_h\}) + \\ & \sum_{T_h \in \mathcal{T}_h} \int_{T_h} \frac{1}{2} (\nabla \cdot w_h)(u_h \cdot v_h) - \sum_{F_h \in \mathcal{F}_h} \int_{F_h} \frac{1}{2} ([\![w_h]\!] \cdot \mathrm{n}_{F_h}) \{u_h \cdot v_h\} \end{aligned} \tag{5.12}$$

The non linear formulation is linearized setting $c((\mathbf{u}, p), (\mathbf{v}, q)) \stackrel{\text{def}}{=} a(\mathbf{u}, \mathbf{v}) + b(p, \mathbf{v}) - b(q, \mathbf{u}) + t_h(\mathbf{u_h^n}, \mathbf{u_h}, \mathbf{v_h}) + t_h(\mathbf{u_h}, \mathbf{u_h^n}, \mathbf{v_h})$ for the jacobian.

This formulation can be compared to its programming counterpart in listings 5.3.

Listing 5.3: C++ implementation of the Navier Stokes problem

```
MeshType Th ;
auto Uh = newCCGSpaceType(Th) ;
auto Uh = newCCGSpace(Th) ;
auto Ph = newP0Space(Th) ;
auto u = *Uh->trial("U",Th::dim) ;
auto v = *Uh->test("V",Th::dim) ;
auto p = *Ph->trial("P") ;
auto q = *Ph->test("Q") ;
FVDomain::algo::MultiIndex<2> _ij(dim,dim) ;
FVDomain::algo::Index _i = _ij.get<0>() ;
FVDomain::algo::Index _j = _ij.get<1>() ;
BilinearForm ah =
  integrate( allCells(Th),
             sum(_i)[ nu*dot(grad(u(_i)),grad(v(_i))) ]
           ) +
  integrate( internalFaces(Th),
             sum(_i)[  -nu*dot(N(),avr(grad(u(_i)))) * jump(v(_i)) -
                       nu*jump(u(_i))*dot(N(),avr(grad(v(_i)))) +
                       eta/H()*jump(u(_i))*jump(v(_i))
                     ]
            ) ;

BilinearForm bh =
  integrate( allCells(Th), -p*div(v) ) +
  integrate( allFaces(Th), avr(p)*dot(N(),jump(v)) ) ) ;

BilinearForm bth =
  integrate( allCells(Th), div(u)*q ) +
  integrate( allFaces(Th), -dot(N(),jump(u)) * avr(q) ) ;

BilinearForm sh =
  integrate( internalFaces(Th), H()*jump(p)*jump(q) ) ;

BilinearForm th1 =
  integrate( allCells(Th),
             sum(_ij)[ (uk(_j)* dxi(_j,u(_i)))*v(_i) ] )  +
  integrate( internalFaces(Th),
             sum(_i)[ (- dot(N(),avr(uk))*jump(u(_i)))*avr(v(_i))] ) +
  integrate( allCells(Th), 0.5 * div(uk)* dot(u,v))  +
  integrate( allFaces(Th),
             (-dot(N(),jump(uk))*dot(avr(u),avr(v)) -
              0.25*dot(N(),jump(uk))* dot(jump(u),jump(v))))   ;

BilinearForm th2 =
  integrate( allCells(Th),
             sum(_ij)[ (dxi(_j,uk(_i))*u(_j))*v(_i) ] ) +
  integrate( internalFaces(Th),
             - dot(N(),avr(u))*dot(jump(uk),avr(v)) ) +
  integrate( allCells(Th), 0.5 * div(u)*dot(uk,v)) +
  integrate( allFaces(Th),
             - dot(N(),jump(u))*avr(dot(uk,v)) ) ;

LinearForm bh =
   integrate( allCells(Th),sum(_i)[ m_f(_i)*v(_i) ] ) ;
```

5.3.2 Kovasznay study case

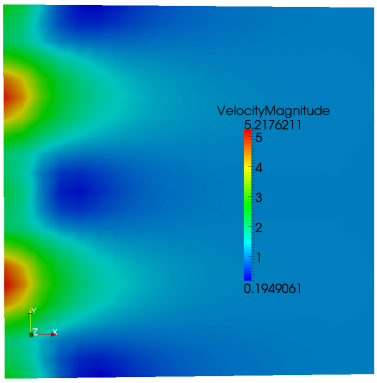
We consider Kovasznay's analytical solution of the navier stokes equations [80] on the square domain $\Omega = (-0.5, 1.5) \times (0, 2)$,

$$u_x = 1 - e^{\pi x}\cos(2\pi y), \qquad u_y = -1/2 e^{\pi x}\sin(2\pi y), \qquad p = -1/2 e^{\pi x}\cos(2\pi y) - \overline{p},$$

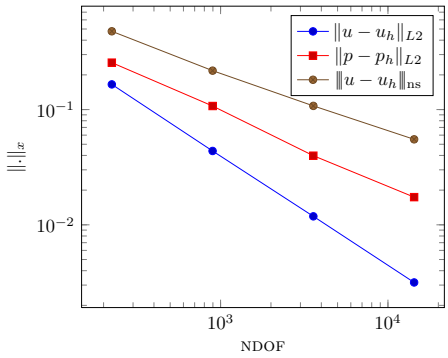
where $\overline{p} = \langle -\frac{1}{2}e^{\pi x}\cos(2\pi y)\rangle_\Omega$ ensures the zero mean constraint for the pressure, $\nu = 3\pi$, and $f = 0$. The example is run on a family of meshes with mesh sizes ranging from 0.5 down to 0.03125. According to Table 1, the errors $\|\!|u - u_h|\!\|_{\text{ns}}$ and $\|p - p_h\|_{L^2(\Omega)}$ converge to first order, while second order is attained for $\|u - u_h\|_{[L^2(\Omega)]^d}$. The results are collected in Table 5.3.2 and convergence curves are plotted in figure 5.7

Table 5.3: Convergence results for the Kovasznay problem

card($\mathcal{T}_h$)	$\|u - u_h\|_{[L^2(\Omega)]^d}$	order	$\|p - p_h\|_{L^2(\Omega)}$	order	$\|\!|u - u_h|\!\|_{\text{ns}}$	order
224	1.6539e-01	–	2.5536e-01	–	4.7777e-01	–
896	4.3732e-02	1.92	1.0737e-01	1.25	2.1759e-01	1.13
3584	1.1847e-02	1.88	3.9802e-02	1.43	1.0763e-01	1.02
14336	3.1620e-03	1.91	1.7385e-02	1.19	5.5182e-02	0.96



(a) 2D view



(b) convergence curves

Figure 5.7: Kovasznay problem

5.3.3 Driven cavity study case

We consider the lid-driven cavity problem on a two-dimensional unit square domain with Dirichlet boundary conditions on all sides, with three stationary sides and one moving side (with velocity tangent to the side) as described in figure 5.8.

We solve the steady incompressible Navier-Stokes problem with the ccG method and present its solution at Reynolds number $Re = \frac{U}{\nu} = 1000$ on a uniform grid of 128×128. In figures 5.11(a), 5.11(c) and 5.11(e) we plot the values of the components of the velocity and its magnitude along axes $x = \frac{1}{2}$ and $y = \frac{1}{2}$ and compare them to some reference results published by Erturk, Corke, and Gökçöl in [63]. We illustrate the results with 2D views in figures 5.10(a), 5.10(b) and 5.9(b). Finally we present the streamlines colored by the velocity magnitude in figure 5.9(a).

Theses figures show that the results are globally correct. Nevertheless, a closer look up of the curves comparing the results to reference solution show that there remains accuracy problems in some difficult regions closed to the boundaries. We are still investigating these problems. We do

not know whether it is due to mesh effects or to non linear solver problems.

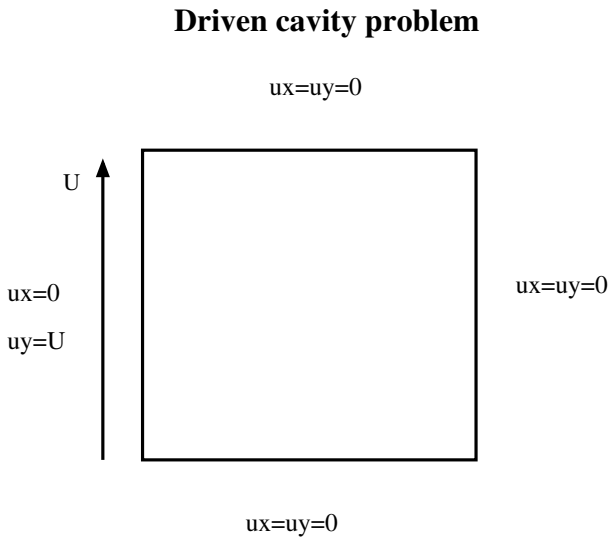
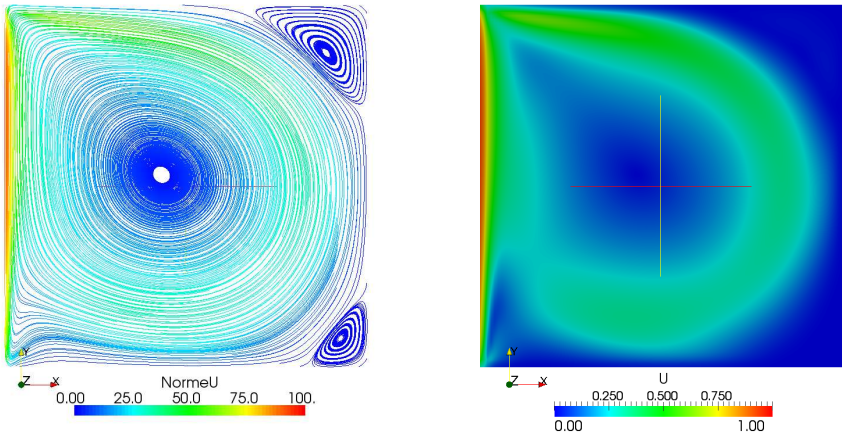


Figure 5.8: Driven cavity problem

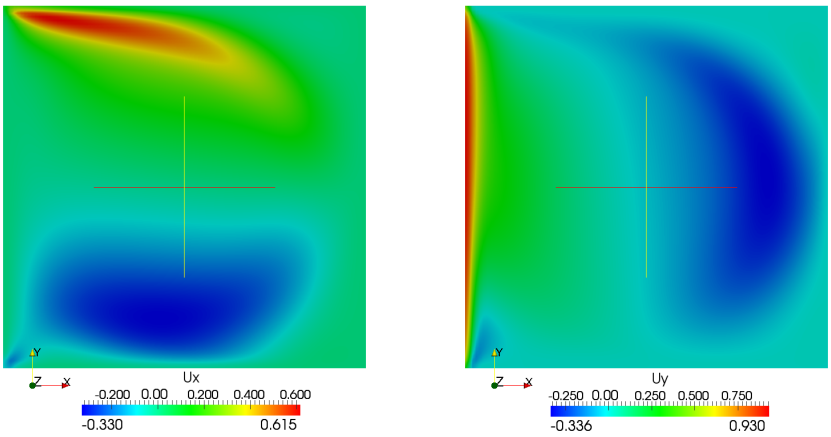


(a) U (b) U

Figure 5.9: Driven cavity problem: Streamline and velocity magnitude

5.4 SPE10 test case

The SPE10 study case is based on data taken from the second model of the 10^{th} SPE test case [45]. The geological model is a $1200 \times 2200 \times 170\, ft$ block discretized with a regular Cartesian grid with $60 \times 220 \times 85$ cells. This model is a part of a Brent sequence. The first 35 top layers represent the Tarbert formation with a prograding near-shore environment (figure 5.16(a)) whereas the lower part corresponds to the Upper Ness formation which is fluvial. The maps of porosity, horizontal



(a) UX (b) UX

Figure 5.10: Driven cavity: velocity components Ux and Uy

and vertical permeability can be downloaded from the web site of the project [17].

5.4.1 Problem settings

Let Ω be the domain represented by the grid, $\partial\Omega_{xmin}$ and $\partial\Omega_{xmax}$, the left and right boundaries of the layer. We consider the following heterogeneous diffusion model problem :

$$\begin{aligned} -\nabla\cdot(\kappa\nabla u) &= 0 && \text{in } \Omega, \\ u &= P_{min} && \text{on } \partial\Omega_{xmin}, \\ u &= P_{max} && \text{on } \partial\Omega_{xmax}, \\ \partial_n u &= 0 && \text{on } \partial\Omega_n \end{aligned} \tag{5.13}$$

where κ is associated to the map of the horizontal permeability field and with the following boundary conditions:

- $P_{xmin} = 500$ on $\partial\Omega_{xmin}$;
- $P_{xmax} = 1000$ on $\partial\Omega_{xmax}$;
- $\partial_n u = 0$ on $\partial\Omega_n = \partial\Omega \setminus \{\Omega_{xmin} \cup \Omega_{xmax}\}$

5.4.2 Results with the SUSHI method

The discrete formulations of this problem 5.13 have been implemented with the SUSHI-method defined in Chapter 2 as in listing 5.4

Listing 5.4: C++ implementation of a_h^{hyb}

```
MeshType Th; // declare T_h
Real Pmin = 500, Pmax=1000;
auto Uh = newHybridSpace(Th);
auto u = Uh->trial("U");
auto v = Uh->test("V");
```

```
BilinearForm ah_hyb =
 integrate(allFaces(Th),k*dot(grad(u),grad(v));
 ah_hyb += on(boundaryFaces(Th,"xmin"),trace(u)=Pmin) ;
 ah_hyb += on(boundaryFaces(Th,"xmax"),trace(u)=Pmax) ;
```

The study case is run on 1, 2, 4, 8 and 16 cores. We collect t_{init}, t_{buid} and t_{solve} respectively the times in seconds to initialize the problem, to build the linear system and to solve it in table 5.4.2 and in the graphic of figure 5.4.2.

In figure 5.12, we have a 3D view of the permeability field and of the solution of the problem.

5.4.3 Results with the multiscale method

The SPE10 test case was initially built to compare upscaling methods since the reservoir which is considered here, is made of two block units with different geological environments. It has been also used to compare multiscale methods in [79].

The Hybrid Multiscale Method described in §2.5.3 is tested on a 2D version of the SPE10 study case. We solve the diffusion problem 2.17 on the first layer of the SPE10 domain discretized by a fine mesh with 65x220x1 cells. We consider the agglomeration leading to a coarse mesh with 10x10x1 cells, and the following boundary conditions:

- $P_{ymin} = 500$ on $\partial\Omega_{ymin}$;
- $P_{ymax} = 1000$ on $\partial\Omega_{ymax}$;
- $\partial_n u = 0$ on $\partial\Omega \setminus \{\Omega_{ymin} \cup \Omega_{ymax}\}$

In figures 5.17 we compare the pressure solution of the hybrid method on the fine mesh to the solution of the multiscale method. In figures 5.18 we compare the velocity solution of the hybrid method on the fine mesh to the solution of the multiscale method.

The test case has been run with three versions of the runtime system presented in Chapter 4, the standard version, the multi-core version based on threads and a mono-core mono-gpu version. We have tested the following hardware configurations:

- `1 CPU` corresponding to a run on 1 core with the standard version;
- `1 TH`, `2 TH`, `4 TH`, `8 TH` corresponding to a run on 1, 2, 4 and 8 cores with the multi-thread version based on the `TBBDriver`;
- `1 GPU` corresponding to a run with 1 core and 1 GPU.

We have monitored the following execution times in seconds:

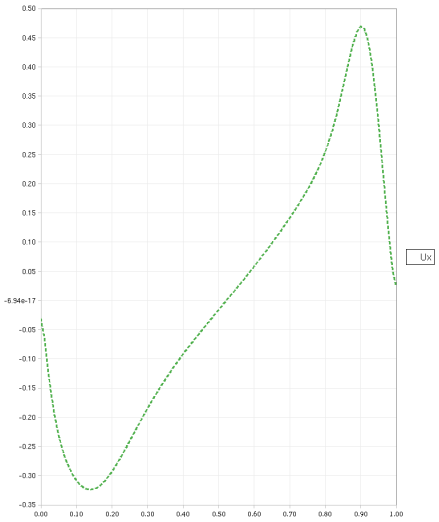
- t_{basis}, the time to compute the basis functions;
- $t_{assembly}$, the time to assemble the coarse system;
- t_{solve}, the time to solve the coarse system;
- $t_{downscale}$, the time to compute the solution on the fine grid.

For comparison, we ran the test case with the SUSHI method on the fine grid, and monitored $t_{fassembly}$ and t_{fsolve} the times to assemble and solve the fine linear system. Performance results are listed in table 5.15(a). The performance of the different configurations are compared in the graphic 5.15(b).

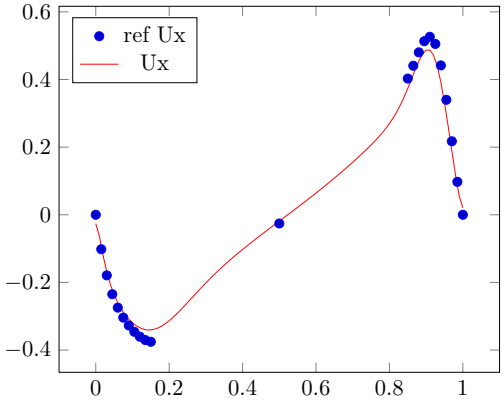
The analysis of the results shows how the computation performance can be enhanced by using all the cores of a processor, or by using a GP-GPU. In the multi-thread version the basis computation is completely parallelized while in the GPU version, only the phase to solve the basis linear

systems is executed on CPU. The phase to assemble basis linear systems is executed on 1 core. That explains why the run on 8 cores has better performance than the run on 1 core and 1 GPU. Further work will be done to use the 8 cores and the GPU.

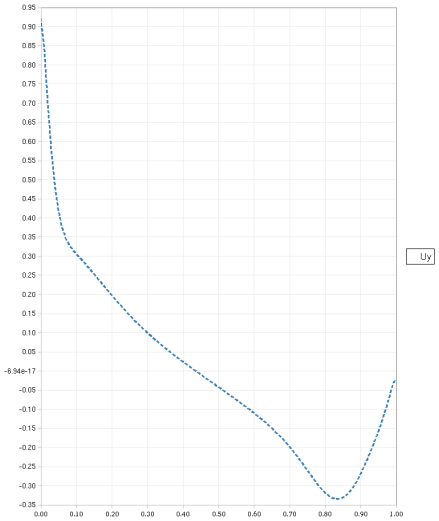
To compare the performance of the multiscale method to the SUSHI method, we have to remember that in an iterative run with several time steps, the basis functions are computed once and only updated if needed. The interest of the multiscale method can be understood by comparing $t_{assembly} + t_{solve} + t_{downscale}$ to $t_{fassembly} + t_{fsolve}$. In an iterative run, to improve performance while preserving the accuracy of the solution, the crucial issue is to compensate the overhead of the basis computation by optimizing the frequency of basis function updates.



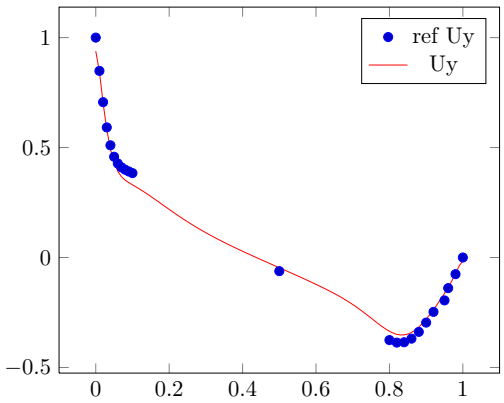
(a) Ux along axe x=0.5



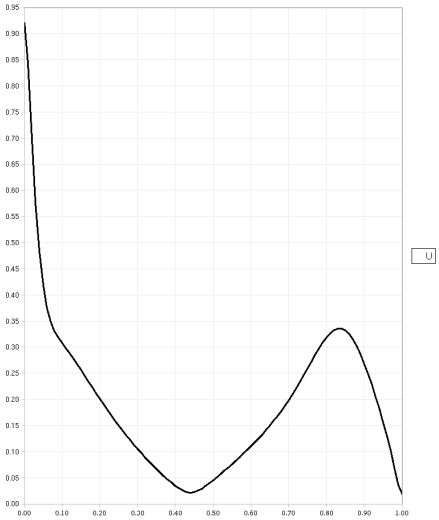
(b) Ux comparaison



(c) Uy curve axe y=0.5

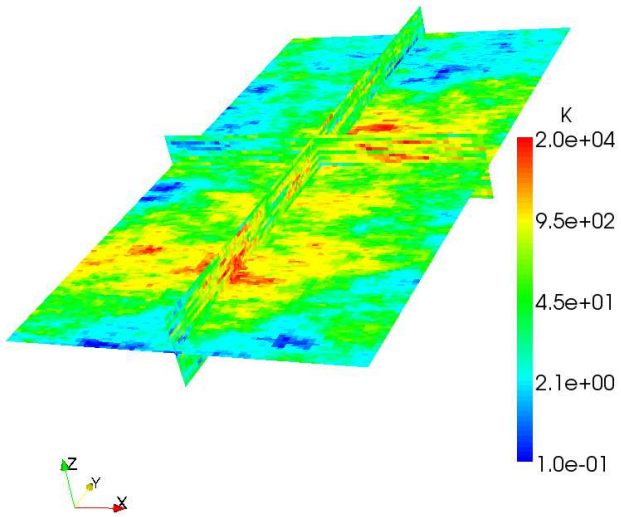


(d) X velocity component

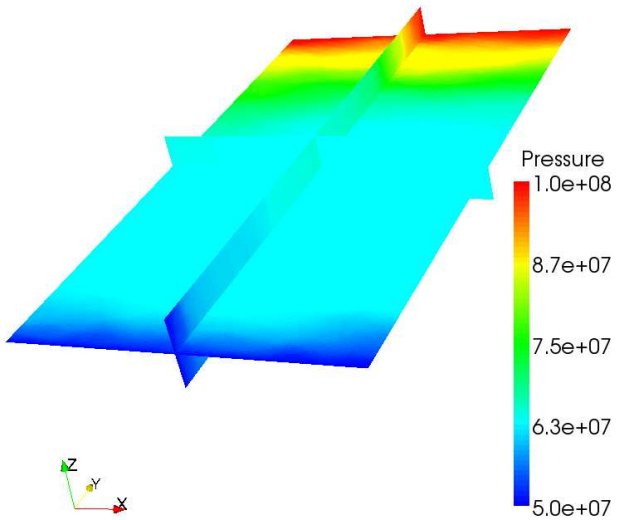


(e) U magnitude along y=0.5

Figure 5.11: Driven cavity: components and magnitude velocity along middle axes



(a) 3D view



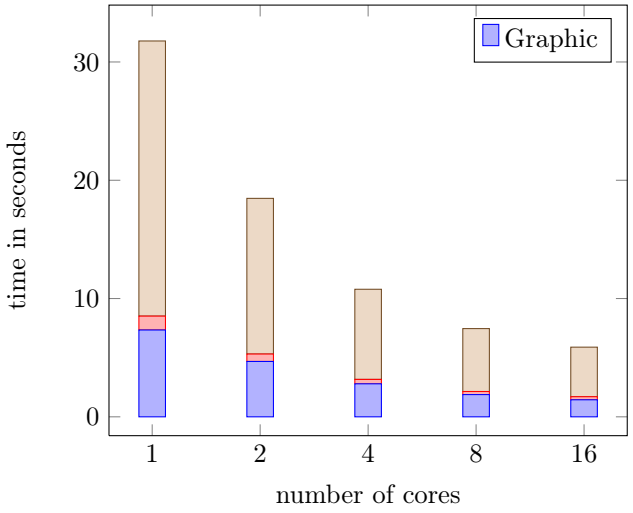
(b) 3D view

Figure 5.12: SPE10 permeability field and pressure solution

NCPU	1	2	4	8	16
t_{init}	7.34	4.68	2.79	1.88	1.44
t_{build}	1.17	0.62	0.37	0.26	0.26
t_{solve}	23.26	13.16	7.62	5.32	4.19

(a) Table

Figure 5.13: SPE10 3D: Performance results of the Hybrid method

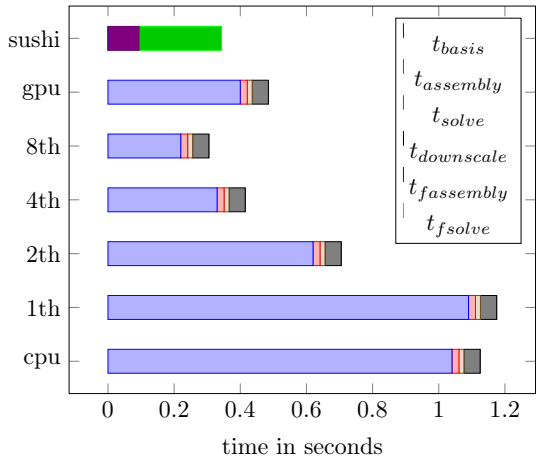


(a) Graphic

Figure 5.14: SPE10 3D: Performance results of the Hybrid method

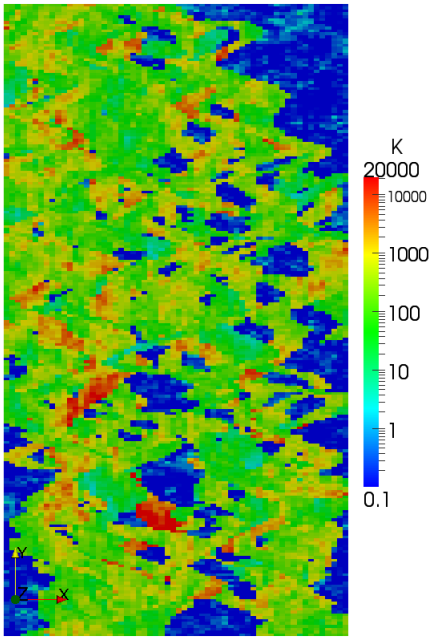
	1 CPU	1 TH	2 TH	4 TH	8 TH	1 GPU
t_{basis}	1.04	1.09	0.62	0.33	0.22	0.40
$t_{assembly}$	0.021	0.021	0.021	0.021	0.021	0.021
t_{solve}	0.015	0.015	0.015	0.015	0.015	0.015
$t_{downscale}$	0.049	0.049	0.049	0.049	0.049	0.049

(a) Table



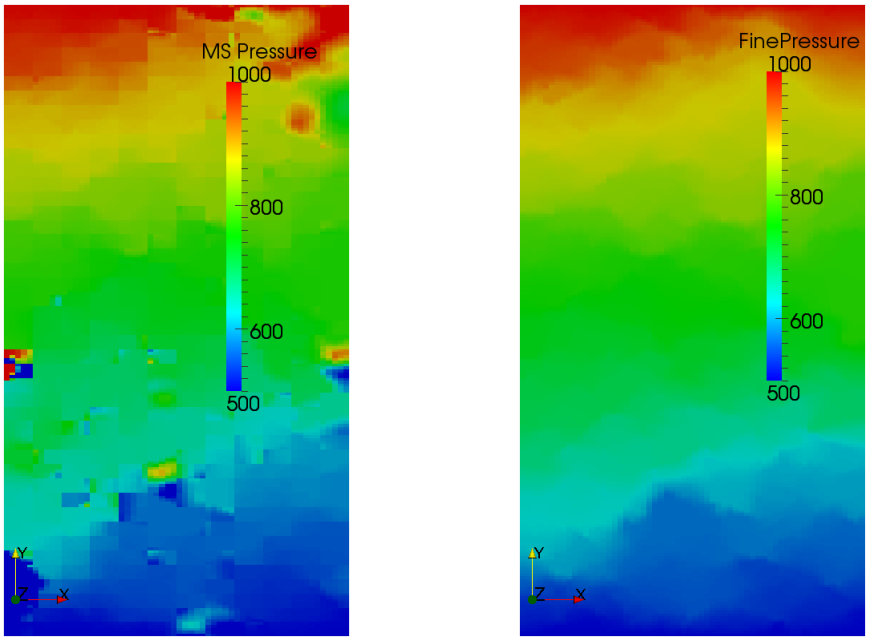
(b) GPU

Figure 5.15: SPE10 2D: Performance results



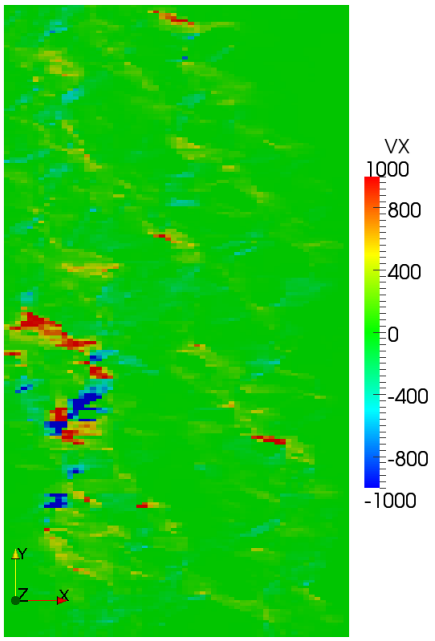
(a) Permeability

Figure 5.16: Multiscale and fine pressure

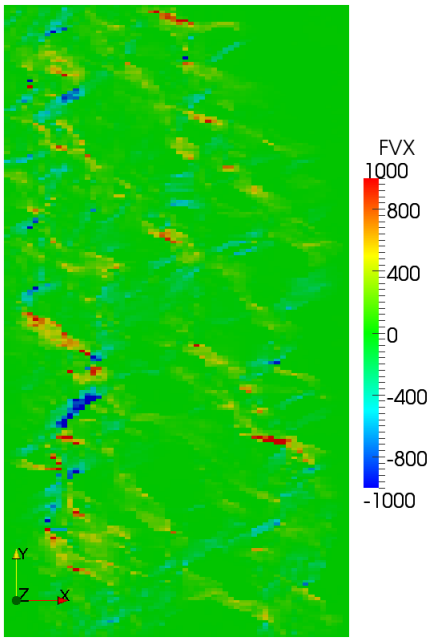


(a) Multiscale P (b) FineHybrid P

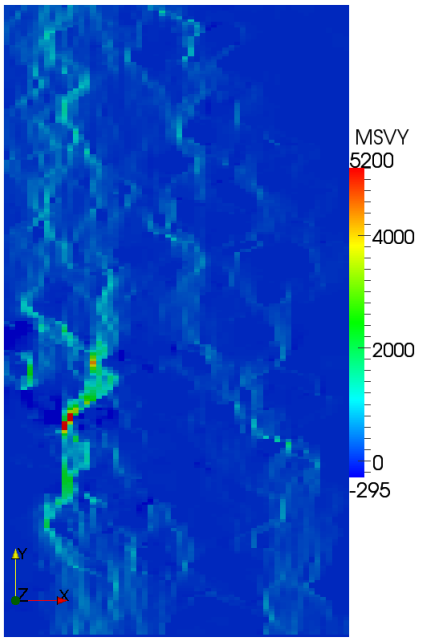
Figure 5.17: Multiscale and fine pressure



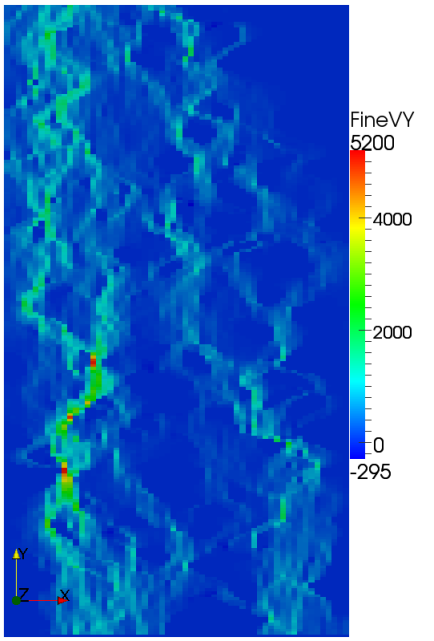
(a) Multiscale VX



(b) SUSHI VX



(c) Multiscale VY



(d) SUSHI VY

Figure 5.18: Multiscale and fine velocity

Chapter 6

Conclusion and perspectives

Scientific computing has to deal with:

(i) the *Modeling* complexity of physical phenomena expressed in terms of systems of Partial Differential Equations;

(ii) the *Numerical* complexity of the discretization methods used to convert PDE problems into systems of algebraic equations;

(iii) the complexity of the algebraic algorithms to solve linear systems on which rely a great part of the performance of applications;

(iv) the complexity of new hardware architectures which provide a high level of parallelism by the mean of heterogeneous memory and computation units;

DS(E)Ls have become an established means to break these different levels of complexity allowing each contributor at each level to focus on a specific aspect of the problem without being hindered by the interaction with the other levels. Up to now this approach was limited to frameworks for linear algebra algorithms or for Finite Element methods for which a unified mathematical formalism has been existing for a long time.

With the emergence of a new consistent unified mathematical frame allowing a unified description of a large family of lowest-order methods, we have been able, as for FE methods, to design a high level language inspired from the mathematical notation, that allows to describe and implement various lowest-order methods. We have designed this language with the `Boost.Proto` library, a powerful framework for DSEL in `C++`, that provides useful tools to define a specific domain language and its grammar, to parse and introspect expressions of its domain and finally to generate algorithms by evaluating and transforming these expressions. Various non trivial academic problems have been solved and different numerical methods have been implemented with the designed DSEL. The analysis of the performance results shows that the overhead of the language is not important compared to standard hand written codes, while the flexibility of the language enables fast prototyping and easy numerical comparisons between different methods to solve a given problem.

The DSEL has been extended to handle multiscale methods. We have at the occasion designed an abstract runtime system layer to address seamlessly heterogeneous hardware architecture, on which relies our generative framework. We have illustrated in that way the capability of our approach to handle both numerical and hardware complexity thanks to the clear separation between the numerical layer with the high level language and the hardware layer thanks to the abstract runtime system model.

In some future works, we plan to extend our DSEL: (i) to take into account the non linear formulation hiding the complexities of derivatives computation introducing the Fréchet derivatives;
(ii) to handle lowest order methods for hyperbolic problems;
(iii) to address new business applications like the linear elasticity, the porous mechanic and advection problems.

The approach studied in this work is one of various approaches that enable to manage the different levels of complexity in scientific computing. Other approaches have been studied to deal with the complexity of software environments, the connexion of scientific applications to data bases, to external workflow tools that launch them as black boxes within business loops. In this context a generative framework based on plugins of the eclipse RCP framework (EMF, Acceleo,...) has been developed. This work is the object of the article "Migration to Model Driven Engineering in the Development Process of Distributed Scientific Application Software" which has been presented at the SPLASH 2012 conference, in Tucson, Arizona.

Appendix A

Sujet de thèse

La spécificité des logiciels scientifiques développés par IFP Energies nouvelles tient avant tout à l'originalité des modèles représentant les situations physiques exprimés sous forme de systèmes d'EDPs assortis de lois de fermeture complexes. Le développement de ces logiciels, conçus pour être exécutés sur les super calculateurs parallèles modernes, nécessite de combiner des méthodes volumes finis robustes et efficaces avec des technologies informatiques qui permettent de tirer au mieux parti de ces calculateurs (parallélisme, gestion de la mémoire, réseaux d'interconnexion, etc). Ces technologies de plus en plus sophistiquées ne peuvent plus être maîtrisées dans leur ensemble par les chercheurs métiers chargés d'implémenter des nouveaux modèles. A ce propos, IFP Energies nouvelles a signé un accord de coopération avec le CEA pour développer et utiliser la plateforme Arcane. Ce choix n'est cependant qu'une reponse partielle au problème posé par le développement, la maintenance ou encore la pérennisation des codes de calcul à IFP Energies nouvelles.

Le but de cette thèse est de compléter l'apport de la plateforme Arcane en proposant des nouveaux outils qui (i) permettront de simplifier le passage du modèle physique à sa résolution numérique en s'appuyant à la fois sur des outils informatiques et un cadre mathématique robuste; (ii) seront intégrés eux-mêmes à la plateforme. Tout d'abord, la programmation générative, l'ingénierie des composants et les langages spécifiques aux domaines d'applications (DSL ou DSEL) sont des technologies clé pour automatiser le développement de programmes. Ces paradigmes permettent d'écrire des codes à la fois lisibles, efficaces et facilement modifiables. Leur application au calcul scientifique était jusqu'à maintenant restreinte aux méthodes de type éléments finis, pour lesquelles un formalisme mathématique unifié existe depuis plus longtemps. L'émergence d'une vision unifiée des méthodes volumes finis et éléments finis [51, 52, 53] permet désormais d'éteindre ces technologies aux approches volumes finis.

Cette thèse se propose donc de développer un langage spécifique aux méthodes de discrétisation Volumes Finis permettant le prototypage rapide de codes industriels ou de recherche. Ce langage sera ensuite intégré à la plate-forme Arcane. Les axes de recherche principaux de ce projet sont (i) l'adaptation du cadre mathématique aux applications d'intérêt pour IFP Energies nouvelles; (ii) le développement d'un langage spécifique permettant de décrire de manière exhaustive ces applications; (iii) le développement d'un interpréteur sous la plateforme Arcane. Le choix entre DSL et DSEL sera évalué pour assurer l'efficacité des applications générées; (iv) finalement, la validation des travaux se fera sur des problèmes académiques puis par le prototypage d'une application industrielle dans le cadre de l'axe "CO2 maîtrisé".

Appendix B

Publications

B.1 Published articles

- "Basic concepts to design a DSL for parallel finite volume applications" : Daniele A. Di Pietro, Jean-Marc Gratien, Florian Häberlein,Anthony Michel, Christophe Prud'homme Proceeding POOSC '09 Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing ACM New York, NY, USA ©2009 table of contents ISBN: 978-1-60558-547-5 doi>10.1145/1595655.1595658
- "Lowest order methods for diffusive problems on general meshes: A unified approach to definition and implementation", Di Pietro and J-M. Gratien, Sixth International Symposium on Finite Volumes for Complex Applications, http://hal.archives-ouvertes.fr/hal-00562500/fr/
- "A Domain Specific Embedded Language in C++ for lowest-order methods for diffusive problem on general meshes", J-M. Gratien, Di Pietro and Christophe Prud'homme, BIT Numerical Mathematics, 53 (1):111-152, 2013, DOI:10.1007/s10543-012-0403-3 - http://hal.archives-ouvertes.fr/hal-00654406
- "Implementing Lowest-Order Methods for Diffusive Problems with a DSEL", J-M. Gratien, Modelling and Simulation in Fluid Dynamics in Porous Media,Springer Proceedings in Mathematics
- "Implementing a Domain Specific Embedded Language for lowest-order variational methods with Boost Proto", J-M. Gratien, CppNow 2012, `https://github.com/boostcon/cppnow_presentations\_2012/blob/master/papers/gratien.pdf`
- "Migration to Model Driven Engineering in the Development Process of Distributed Scientific Application Software",R. Gayno, J-M. Gratien, D. Rahon, S. Schneider, G.Lefur, proceedings of SPLASH 2012 conference, in Tucson, Arizona

B.2 Conference presentations

- POOSC 2009, Genova, Italia
- Fluid Dynamics in Porous Media, 2010, Coimbra, Portugal
- CppNow 2012, Aspen, Colorado, USA
- ECCOMAS, 2012, Vienna, Austria

Appendix C

Overview of multiscale methods in geoscience

Multiscale methods have been introduced to solve PDE systems modeling phenomena that are governed by physical processes occurring on a wide range of time and/or length scales. This kind of problems cannot be solved on the finest grid due to time and memory limitations. They consist in incorporating fine-scale information into a set of coarse-scale equations in a way that is consistent with the local properties of the mathematical model on the unresolved subscale(s). In geoscience, multiscale methods are considered for the simulation of pressure and (phase) velocities in porous media flow. Multiscale behavior in porous media flow are due to heterogeneities in rock and sand formations which are reflected in the permeability tensor used in the governing partial differential equations. To accurately resolve the pressure distribution, it is necessary to account for the influence of fine-scale variations in the coefficients of the permeability tensor. For incompressible flow, the pressure equation reduces to the following variable coefficient Poisson equation:

$$\begin{aligned} v &= -\kappa \nabla p \text{ on } \Omega, \\ \nabla \cdot v &= q \text{ on } \Omega, \\ p &= g \text{ on } \partial\Omega_D, \\ \partial_n p &= 0 \text{ on } \partial\Omega_N = \partial\Omega \setminus \partial\Omega_D, \end{aligned} \tag{C.1}$$

where

- q is a source term,
- κ stands for the symmetric positive definite permeability tensor that typically has a multiscale structure due to the strongly heterogeneous nature of natural porous media,
- $\partial\Omega_D$ (respectively $\partial\Omega_N$) is a subset of the boundary $\partial\Omega$ of Ω where Dirichlet (respectively Neumann) boundary conditions are enforced.

Kippe V., Aarnes J. E. and Lie K. A. have done an interesting overview on multiscale methods in [91]. They say:

> "The literature on numerical methods for elliptic problems contains a number of multiscale methods that are geared toward solving problems with highly oscillatory coefficients. Examples include the multiscale finite-element method (MsFEM) [73], the variational multiscale method [76], the mixed multiscale finite-element method (MxMsFEM)[44], and the multiscale finite-volume method (MsFVM)[75]. All of these methods are based on a hierarchical two-scale approach, where the general idea is to derive a set of equations on a coarse scale that embodies the impact of subgrid variations in the elliptic

coefficients. To this end, subgrid computations are performed as part of the multiscale method to estimate how these fine-scale variations influence the coarse-grid solution."

In this section inspired from Kippe V and al. work [91] we present the MsFEM, MxMsFEM and MsFVM methods. All these methods are based on a coarse and a fine grid. For the following section we introduce a few notations: we denote $\mathcal{T}_h$ and $\mathcal{T}_H$ respectively the fine and the coarse grid. The exponent ^{c} and ^{f} are used for respectively elements related to the coarse grid and the fine grid. We denote τ^c and σ^c respectively cell and face elements of $\mathcal{T}_H$ and τ^f and σ^f, cell and face elements of $\mathcal{T}_h$.

C.1 Multiscale Finite-Element methods

MsFEMs are based on multiscale basis functions and on a global numerical formulation that couples them. Basis functions are designed to capture the multiscale features of the solution. Important multiscale features of the solution are incorporated into these localized basis functions which contain information about the scales that are smaller as well as larger than the local numerical scale defined by the basis functions. A global formulation couples these basis functions to provide an accurate approximation of the solution.

Basis functions Let $\mathcal{T}_H$ be the coarse grid, a partition of Ω into finite elements. We assume that the coarse grid can be resolved via a finer resolution $\mathcal{T}_h$ called the fine grid. Let $\mathbf{x}_i$ be the interior nodes of the mesh $\mathcal{T}_H$ and ϕ_i^0 be the nodal basis of the standard finite element space $W_h = \text{span}\{\phi_i^0\}$. If $\mathcal{T}_H$ is a triangular partition, we denote by $S_i = \text{supp}(\phi_i^0)$ (the support of ϕ_i^0) and define ϕ_i with support in S_i as follows : $\forall \tau^c \in , \tau^c \subset S_i$

$$\begin{aligned} -\nabla{\cdot}\kappa\nabla\phi_i &= 0 \text{ on } \tau^c, \\ \phi_i &= \phi_i^0 \text{ on } \partial\tau^c, \end{aligned} \tag{C.2}$$

Those multiscale basis functions coincide with standard finite element basis functions on the boundaries of a coarse-grid block τ^c, and are oscillatory in the interior of each coarse-grid block. In general, one solves (C.2) on the fine grid to compute basis functions. Once the basis functions are constructed, we denote by P_h the discrete space spanned by ϕ_i:

$$P_h = \text{span}\{\phi_i\}$$

.

Global formulation The dimension computation is reduced by representing the fine-scale solution with multiscale basis functions as follows:

$$p_h = \sum_i p_i \phi_i$$

where p_i are the values of the solution at coarse-grid nodal points and substituing p_h in the fine-scale equation. To obtain the coarse-level equation, we multiply then the resulting equation with coarse-scale test functions. In the case of Galerkin finite element methods with conforming basis functions, the MsFEM reads : find $p_h \in P_h$ such that

$$\int_\Omega \kappa\nabla p_h \cdot \nabla v_h = \int_\Omega f v_h \qquad \forall v_h \in P_h \tag{C.3}$$

When the test functions are chosen from W_h we obtain the Petrov-Galerkin version of the MsFEM which reads: find $p_h \in P_h$ such that

$$\int_\Omega \kappa\nabla p_h \cdot \nabla v_h = \int_\Omega f v_h \qquad \forall v_h \in W_h \tag{C.4}$$

Equation (C.3) or (C.4) couples the multiscale basis functions. The values of the solution at the nodes of the coarse-grid block are obtained solving the coarse linear system of equations, determining the coarse solution.

C.2 Multiscale Finite-Volume method

The MsFVM is based on a finite-volume formulation. In a finite-volume method, a family of control volumes is introduced and mass conservation is imposed locally on each control volume τ:

$$\int_{\partial\tau} -\kappa\nabla P \cdot \mathrm{n} = \int_{\tau} q \tag{C.5}$$

where n is the outward unit normal on $\partial\tau$. Usually a two-point flux approximation scheme is used as finite-volume method expressing the flux accross an interface $\sigma = \partial\tau_i \cap \partial\tau_j$ as:

$$\int_{\sigma} -\kappa\nabla P \cdot \mathrm{n} = T_\sigma(p_i - p_j)$$

where T_σ is the transmibility coefficient related to σ of the two-point scheme. The method, introduced by Jenny and al. [77, 21] is a control-volume finite-element formulation on the coarse mesh $\mathcal{T}_H$. The pressure $P = \sum_i p_i\phi_i$ is expressed as a linear combination of basis functions and substitued in (C.5). Thus, we obtain $\forall\tau_i^c \subset \mathcal{T}_H$:

$$\int_{\partial\tau_i^c} \mathrm{n} \cdot \kappa\nabla(\sum_j p_j\phi_j) = -\sum_j p_j \int_{\partial\tau_i} \mathrm{n} \cdot \kappa\nabla\phi_j = -\sum_j p_j f_{i,j} = \sum_i \int_{\tau_i} q \tag{C.6}$$

The quantitiy $f_{i,j}$ denote the flux over the boundary of cell τ_i^c due to the basis function centered in cell τ_j^c. These quantities are refered as the MsFVM transmissibilities. Equations (C.6) give a coarse linear system that can be solved for $\{p_j\}$, and the fine-scale pressure solution is given as:

$$P = \sum_j p_j\phi_j$$

The fine-scale pressure solution gives a velocity field that is mass conservative on the coarse mesh $\mathcal{T}_H$, but generally not on the fine mesh $\mathcal{T}_h$. A mass-conservative velocity solution on the fine scale is obtained by solving (C.1) within each control volume $\tau^c \in \mathcal{T}_H$ using the following Neumann boundary condition

$$\mathbf{v} \cdot \mathrm{n} = -\kappa\nabla P \cdot \mathrm{n} \text{ on } \partial\tau^c$$

with $P = \sum_j p_j\phi_j$.

C.3 Mixed Multiscale Finite-Element method

The mixed multiscale finite-element method was first introduced by Chen and Hou in [44] and used for the simulation of two-phase flows later by Aarnes in [20]. In this method, the coarse mesh $\mathcal{T}_H$ is defined coarsening the initial fine grid $\mathcal{T}_h$. Basis functions are related to coarse faces σ^c of $\mathcal{T}_H$. They are computed by solving local problems at the fine scale, then used to build a mixed hybrid linear system on the coarse grid. After the resolution of this coarse system, the basis functions are used to compute the pressures and the velocities on the fine grid from the values of the coarse solution.

For the model problem (C.1), the mixed formulation reads: find $(u, p) \in H_0^{div}(\Omega) \times L^2(\Omega)$ such that,

$$(\kappa^{-1}u, v) - (p, \nabla{\cdot}v) = 0, \forall v \in H_0^{div}(\Omega), \tag{C.7}$$

$$(\nabla{\cdot}u, l) = (q, l), \forall l \in L^2(\Omega). \tag{C.8}$$

where $H_0^{div}(\Omega) = \{v \in L^2(\Omega)^n : \nabla{\cdot}v \in L^2(\Omega), v \cdot \mathrm{n} = 0 \text{ on } \partial\Omega\}$.

The mixed finite-element methods consist in searching an discrete solution (u_h, p_h) of (C.7) that is confined to lie in finite-dimensional subspaces $\mathcal{V} \in H_0^{div}(\Omega)$ and $\mathcal{W} \in L^2(\Omega)$. The discrete formulation reads:
find $(u_h, p_h) \in V \times W$ such that,

$$\begin{aligned}(\kappa^{-1}u_h, v_h) - (p_h, \nabla{\cdot}v_h) = 0, \forall v_h \in V,\\ (\nabla{\cdot}u_h, l_h) = (q_h, l_h), \forall l_h \in W\end{aligned} \tag{C.9}$$

There are two standard mixed finite element methods.

- the lowest-order Raviart-Thomas method (RT0) [88] consists in setting:

$$W = P_0(\mathcal{T}_h) \text{ and } V \subset \{v \subset H^{div}(\Omega) : \nabla{\cdot}v \in W, v \cdot \mathrm{n} \in P_0(\sigma^c)\}$$

  where $\sigma^c = \partial\tau_i^c \cap \partial\tau_j^c$ for $\tau_i^c \in \mathcal{T}_H, \tau_j^c \in \mathcal{T}_H$ and n is a uniquely oriented unit normal. The normal component of the velocity on each interface is constant.

- the lowest-order Brezzi-Douglas-Marini method (BDM1) [43] consists in setting:

$$W = P_0(\mathcal{T}_h) \text{ and } V \subset \{v \subset H^{div}(\Omega) : \nabla{\cdot}v \in W, v \cdot \mathrm{n} \in P_1(\sigma^c)\}$$

  In this case the normal component of the velocity on each interface is linear.

The velocity approximation spaces in mixed methods are spanned by the basis functions associated with interfaces in the mesh. For RT0, there is only one degree of freedom per interface and basis functions represent flow units across element interfaces.

Hou and Wu introduced in [74] a new family of multiscale finite-element methods. These methods gives mass-conservative velocity fields on the coarse mesh and also on the fine mesh in coarse blocks not containing sources. On the coarse scale, they generalize the standard RT0 method with the piecewise linear velocity basis functions accountting for subgrid variations in the coefficients. The basis functions are related to a coarse interface $\sigma^c = \partial\tau_i^c \cap \partial\tau_j^c$. They are the solutions of (C.1) restricted to $\tau_i^c \cup \tau_j^c$ with source terms specified in such a way that unit flow is forced across σ^c. The multiscale velocity basis functions $\psi_{i,j}$ are defined as follows:
$\mathrm{supp}(\psi_{i,j}) = \tau_i \cup \tau_j$ and $\psi_{i,j}$ is solution of

$$\begin{cases} \psi_{i,j} = -\kappa\nabla\phi_{i,j}, \\ -\nabla \cdot (\kappa\nabla\phi_{i,j}) = w_1 \text{ on } \tau_i^c, \\ -\nabla \cdot (\kappa\nabla\phi_{i,j}) = -w_2 \text{ on}\tau_j^c, \\ \kappa\nabla\phi_{i,j} \cdot \mathrm{n} = 0 \text{ on } \partial(\tau_i^c \cup \tau_j^c) \end{cases} \tag{C.10}$$

where $w_{i,i\in\{1,2\}}$ are weight functions defined by

$$w_i = \frac{\mathrm{trace}(\kappa)}{\int_{\tau_i} \mathrm{trace}(\kappa)} \text{ if } q|_{\tau_i} = 0, w_i = q \text{ otherwise .} \tag{C.11}$$

The functions $\psi_{i,j}$ are thus defined to set a unit flux on σ^c and no fluxes on $\partial(\tau_i^c \cup \tau_j^c)$. The term $\mathrm{trace}(\kappa)$ in the weight functions or in the boundary conditions of (C.10) is the trace of the permeability tensor κ. It enables to weight the fine fluxes at the fine scale according to the permeability field.

The coarse scale approximation consists in searching a solution in the discrete approximation spaces $(V^{ms}, P^0(\mathcal{T}_h))$ where $V^{ms} = \mathrm{span}\{\psi_{i,j}\}$

Bibliography

[1] Charm++ Web page. http://charm.cs.uiuc.edu/.

[2] CUDA Web page. http://www.nvidia.com/content/cudazone/cuda_sdk/Linear_Algebra.html.

[3] CULA Web page. http://www.culatools.com/.

[4] DUNE Web page. http://www.dune-project.org/.

[5] EIGEN Web page. http://eigen.tuxfamily.org/index.php?title=Main_Page.

[6] FENics Web page. http://fenicsproject.org/.

[7] FENics Wikipedia page. http://en.wikipedia.org/wiki/FEniCS_Project.

[8] GetDP Web page. http://geuz.org/getdp/.

[9] GetFEM++ Web page. http://download.gna.org/getfem/html/homepage/.

[10] HWLOC Web page. http://www.open-mpi.org/projects/hwloc/.

[11] MTL4 Web page. http://www.simunova.com/de/node/24.

[12] NT2 Web page. http://nt2.lri.fr/.

[13] Phoenix Web page. www.boost.org/libs/spirit/phoenix/.

[14] Quaff Web page. http://sourceforge.net/projects/quaff/.

[15] Spirit Web page. http://boost-spirit.com/home/.

[16] Sundance Web page. http://www.math.ttu.edu/~klong/sundance/html/index.html.

[17] Web site for the 10th spe comparative solution project: www.spe.org/csp/.

[18] XKaapi Web page. http://kaapi.gforge.inria.fr/dokuwiki/doku.php.

[19] ISO/IEC 14977, 1996(E).

[20] J.E. Aarnes. On the use of a mixed multiscale finite element method for greater flexibility and increased speed or improved accuracy in reservoir simulation. *Multiscale Model. Simul.*, 2(3):421–439, 2004.

[21] J.E. Aarnes, S. Krogstad, and K.-A. Lie. Multiscale mixed/mimetic methods on corner-point grids. *Comput. Geosci.*, 12(3):297–315, 2008.

[22] I. Aavatsmark, T. Barkve, Ø. Bøe, and T. Mannseth. Discretization on non-orthogonal, curvilinear grids for multi-phase flow. In *Prov. of the 4th European Conf. on the Mathematics of Oil Recovery*, volume D, Røros, Norway, 1994.

[23] I. Aavatsmark, T. Barkve, Ø. Bøe, and T. Mannseth. Discretization on non-orthogonal, quadrilateral grids for inhomogeneous, anisotropic media. *J. Comput. Phys.*, 127:2–14, 1996.

[24] I. Aavatsmark, T. Barkve, Ø. Bøe, and T. Mannseth. Discretization on unstructured grids for inhomogeneous, anisotropic media, Part I: Derivation of the methods. *SIAM J. Sci. Comput.*, 19(5):1700–1716, 1998.

[25] I. Aavatsmark, T. Barkve, Ø. Bøe, and T. Mannseth. Discretization on unstructured grids for inhomogeneous, anisotropic media, Part II: Discussion and numerical results. *SIAM J. Sci. Comput.*, 19(5):1717–1736, 1998.

[26] I. Aavatsmark, G. T. Eigestad, B. T. Mallison, and J. M. Nordbotten. A compact multipoint flux approximation method with improved robustness. *Numer. Methods Partial Differ. Eq.*, 24:1329–1360, 2008.

[27] I. Aavatsmark, G. T. Eigestad, B. T. Mallison, and J. M. Nordbotten. A compact multipoint flux approximation method with improved robustness. *Numer. Methods Partial Differ. Eq.*, 24:1329–1360, 2008.

[28] Davis Abrahams and Leksey Gurtovoy. C++ template metaprogramming : Concepts, tools, and techniques from boost and beyond. C++ in Depth Series. Addison-Wesley Professional, 2004.

[29] L. Agélas, D. A. Di Pietro, and J. Droniou. The G method for heterogeneous anisotropic diffusion on general meshes. *M2AN Math. Model. Numer. Anal.*, 44(4):597–625, 2010. DOI: 10.1051/m2an/2010021.

[30] L. Agélas, D. A. Di Pietro, and J. Droniou. The G method for heterogeneous anisotropic diffusion on general meshes. *M2AN Math. Model. Numer. Anal.*, 44(4):597–625, 2010.

[31] L. Agélas, D. A Di Pietro, R. Eymard, and R. Masson. An abstract analysis framework for nonconforming approximations of anisotropic heterogeneous diffusion. Preprint available at http://hal.archives-ouvertes.fr/hal-00318390/fr, 2010.

[32] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Mathieu Faverge, Julien Langou, Hatem Ltaief, and Stanimire Tomov. LU Factorization for Accelerator-based Systems. In *9th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA 11)*, Sharm El-Sheikh, Égypte, June 2011.

[33] D. N. Arnold. An interior penalty finite element method with discontinuous elements. *SIAM J. Numer. Anal.*, 19:742–760, 1982.

[34] Pierre Aubert and Nicolas Di Césaré. Expression templates and forward mode automatic differentiation. Computer and information Science, chapter 37, pages 311-315. Springer, New York, NY, 2001.

[35] Cédric Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective.* PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, December 2011.

[36] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[37] S. Balay, J. Brown, K. Buschelman, W. D. Gropp, D. Kaushik, M. G. Knepley, L. Curfman McInnes, B. F. Smith, and H. Zhang. PETSc Web page, 2011. http://www.mcs.anl.gov/petsc.

[38] F. Bassi, L. Botti, A. Colombo, D. A. Di Pietro, and P. Tesini. On the flexibility of agglomeration based physical space discontinuous Galerkin discretizations. *J. Comput. Phys.*, 231(1–2):45–65, 2012.

[39] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[40] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–11, New York, NY, USA, 2009. ACM.

[41] F. Brezzi, K. Lipnikov, and M. Shashkov. Convergence of mimetic finite difference methods for diffusion problems on polyhedral meshes. *SIAM J. Numer. Anal.*, 45:1872–1896, 2005.

[42] F. Brezzi, K. Lipnikov, and V. Simoncini. A family of mimetic finite difference methods on polygonal and polyhedral meshes. *M3AS*, 15:1533–1553, 2005.

[43] Jr. J.D. Marini L.D. Brezzi, F. Two families of mixed elements for second order elliptic problems. *Numer. Math.*, 47:217–235, 1985.

[44] Z. Chen and T.Y. Hou. A mixed multiscale finite element method for elliptic problems with oscillating coefficients. *Mathematics of Computation*, 72(242):541–576, 2002.

[45] M.A. Christie and M.J. Blunt. Tenth spe comparative solution project: A comparison of upscaling techniques. In *SPE Reservoir Simulation Symposium, Houston, Texas, 11–14 February 2001.*

[46] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems.* North Holland, Amsterdam, 1978.

[47] P. G. Ciarlet. Basic error estimates for elliptic problems. In P. G. Ciarlet and J.-L-Lions, editors, *Handbook of Numerical Analysis*, volume II: Finite Element Methods, chapter 2. North-Holland, Amsterdam, 1991.

[48] C. Prud'homme Di Pietro, J-M. Gratien. A domain specific embedded language in c++ for lowest-order methods for diffusive problem on general meshes. *BIT Numerical Mathematics*, 53:111–152, 2013.

[49] D. A. Di Pietro. Cell-centered Galerkin methods. *C. R. Math. Acad. Sci. Paris*, 348:31–34, 2010.

[50] D. A. Di Pietro. Cell centered Galerkin methods. *C. R. Acad. Sci. Paris, Ser. I*, 348:31–34, 2010. DOI: 10.1016/j.crma.2009.11.012.

[51] D. A. Di Pietro. Cell centered galerkin methods for diffusive problems. *M2AN Math. Model. Numer. Anal.*, 2010. Published online.

[52] D. A. Di Pietro. A compact cell-centered Galerkin method with subgrid stabilization. *C. R. Acad. Sci. Paris, Ser. I.*, 348(1–2):93–98, 2011.

[53] D. A. Di Pietro. Cell centered Galerkin methods for diffusive problems. *M2AN Math. Model. Numer. Anal.*, 46(1):111–144, 2012.

[54] D. A. Di Pietro and A. Ern. *Mathematical Aspects of Discontinuous Galkerin Methods.* Number 69 in Mathématiques & Applications. Springer Verlag, Berlin, 2011.

[55] D. A. Di Pietro and A. Ern. Analysis of a discontinuous Galerkin method for heterogeneous diffusion problems with low-regularity solutions. *Numer. Methods for Partial Differential Equations*, 28(4):1161–1177, 2012. Published online. DOI: 10.1002/num.20675.

[56] D. A. Di Pietro, A. Ern, and J.-L. Guermond. Discontinuous Galerkin methods for anisotropic semi-definite diffusion with advection. *SIAM J. Numer. Anal.*, 46(2):805–831, 2008.

[57] D. A. Di Pietro and J-M. Gratien. Lowest order methods for diffusive problems on general meshes: A unified approach to definition and implementation. In *FVCA6 proceedings*, 2011.

[58] J. Droniou and R. Eymard. A mixed finite volume scheme for anisotropic diffusion problems on any grid. *Num. Math.*, 105(1):35–71, 2006.

[59] J. Droniou, R. Eymard, T. Gallouët, and R. Herbin. A unified approach to mimetic finite difference, hybrid finite volume and mixed finite volume methods. *M3AS*, 20(2):265–295, 2010.

[60] M.G. Edwards and C.F. Rogers. A flux continuous scheme for the full tensor pressure equation. In *Prov. of the 4th European Conf. on the Mathematics of Oil Recovery*, volume D, Røros, Norway, 1994.

[61] M.G. Edwards and C.F. Rogers. Finite volume discretization with imposed flux continuity for the general tensor pressure equation. *Comput. Geosci.*, 2:259–290, 1998.

[62] Yalchin R. Efendiev, Thomas Y. Hou, and Xiao-Hui Wu. Convergence of a nonconforming multiscale finite element method, 2000.

[63] Corke T. C. Erturk, E. and C. Gökçöl. Numerical solutions of 2-d steady incompressible driven cavity flow at high reynolds numbers. *Int. J. Numer. Meth. Fluids*, 48, 2005.

[64] R. Eymard, Th. Gallouët, and R. Herbin. Discretization of heterogeneous and anisotropic diffusion problems on general nonconforming meshes SUSHI: a scheme using stabilization and hybrid interfaces. *IMA J. Numer. Anal.*, 2010. Published online.

[65] R. Eymard, Th. Gallouët, and R. Herbin. Discretization of heterogeneous and anisotropic diffusion problems on general nonconforming meshes SUSHI: a scheme using stabilization and hybrid interfaces. *IMA J. Numer. Anal.*, 30:1009–1043, 2010.

[66] R. Eymard, G. Henry, R. Herbin, F. Hubert, R. Klofkorn, and G. Manzini. 3D benchmark on discretization schemes for anisotropic diffusion problems on general grids. In J. Halama R. Herbin J. Fovrt, J. Furst and F. Hubert, editors, *Finite Volumes for Complex Applications VI Problems & Perspectives*, pages 95–130. Springer-Verlag, 2011.

[67] C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[68] Jean-Marc Gratien. Implementing a Domain Specific Embedded Language for lowest-order variational methods with Boost Proto. available at `http://hal.archives-ouvertes.fr/hal-00788281`, 2012.

[69] Gilles Grospellier and Benoit Lelandais. The arcane development framework. In *Proceedings of the 8th workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '09, pages 4:1–4:11, New York, NY, USA, 2009. ACM.

[70] F. Hecht and O. Pironneau. *FreeFEM++ Manual*. Laboratoire Jacques Louis Lions, 2005.

[71] R. Herbin and F. Hubert. Benchmark on discretization schemes for anisotropic diffusion problems on general grids. In *Finite Volumes for Complex Applications V*, pages 659–692. John Wiley & Sons, 2008.

[72] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An Overview of Trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[73] T.Y. Hou and X.H. Wu. A multiscale finite element method for elliptic problems in composite materials and porous media. *J. Comput. Phys.*, 134:169–189, 1997.

[74] T.Y. Hou, X.H. Wu, and Z. Cai. Convergence of a multiscale finite element method for elliptic problems with rapidly oscillating coefficients. *Math. Comput.*, 68:913–943, 1999.

[75] Wu X.H. Hou, T.Y. A multiscale finite element method for elliptic problems in composite materials and porous media. *J. Comput. Phys.*, 134(1):169–189, 1997.

[76] Feijoo G. Mazzei L. Quincy J. Hughes, T. The variational multiscale method - a paradigm for computational mechanics. *Comput. Methods Appl. Mech. Eng.*, 166(3-24), 1998.

[77] P. Jenny, S.H. Lee, and H. Tchelepi. An adaptive fully implicit multi-scale finite-volume algorithm for multi-phase flow in porous media. *J. Comp. Phys.*, 217:627–641, 2006.

[78] Hartmut Kaiser. HPX Web page, 2012. `http://stellar.cct.lsu.edu`.

[79] V. Kippe, J.E. Aarnes, and K-A Lie. A comparison of multiscale methods for elliptic problems in porous media flow. *Comput. Geosci.*, 12:377–398, 2008.

[80] L. S. G. Kovasznay. Laminar flow behind a two-dimensional grid. *Proc. Camb. Philos. Soc.*, 44:58–62, 1948.

[81] A. Logg, J. Hoffman, R.C. Kirby, and J. Jansson. Fenics. `http://www.fenics.org/`, 2005.

[82] Kirby R. Long, K. and B. van Bloemen Waanders. Unified embedded parallel finite element computations via software-based fréchet differentiation. *SIAM Journal on Scientific Computing*, 32:3323–3351, 2010.

[83] E. Niebler. *`boost::proto` documentation*, 2011. http://www.boost.org/doc/libs/1_47_0/doc/html/proto.html.

[84] C. Prud'homme. A domain specific embedded language in C++ for automatic differentiation, projection, integration and variational formulations. *Scientific Programming*, 2(14):81–110, 2006.

[85] C. Prud'homme. Life: Overview of a unified c++ implementation of the finite and spectral element methods in 1d, 2d and 3d. In Workshop On State-Of-The-Art In Scientific And Parallel Computing, Lecture Notes in Computer Science, page 10. Springer-Verlag, 2007.

[86] C. Prud'homme, V. Chabannes, V. Doyeux, M. Ismail, A. Samake, and G. Pena. Feel++: A computational framework for Galerkin methods and advanced numerical methods. (hal-00662868), January 2012.

[87] C. Prud'homme, V. Chabannes, G. Pena, and S. Veys. Feel++: Finite Element Embedded Language in C++. Free Software available at `http://www.feelpp.org`. Contributions from A. Samake, V. Doyeux, M. Ismail.

[88] Thomas J.M. Raviart, P.A. A mixed finite element method for 2nd order elliptic problems. in: Mathematical aspects of finite element methods, 1977.

[89] Y. Saad. Iterative methods for sparse linear systems (2nd edition).

[90] Xiao Hui Wu T.Y. Hou and Yu Zheng. Removing the cell resonance error in multiscale finite element method via a petrov galerkine formulation. *COMM. MATH. SCI.*, 2(2):185–205, 2004.

[91] Kippe V., Aarnes J. E., and Lie K. A. A comparison of multiscale methods for elliptic problems in porous media flow. *Computational Geosciences*, 12(3):377–398, 2008.

[92] Todd Veldhuizen. Using c++ template metaprograms. C++ report, 7(4):36-43, May 1995. reprinted in C++ Gems, ed. Stanley Lippman, 1995.